To appear in Formal Aspects of Computing

Blaming the Client: On Data Refinement in the Presence of Pointers

Ivana Filipović[1], Peter O'Hearn[1], Noah Torp-Smith[2] and Hongseok Yang[1]

[1] Queen Mary, University of London, Mile End Road, London, E1 4NS, United Kingdom
[2] Maconomy A/S, Vordingborggade 18-22, DK - 2100 Copenhagen, Denmark

Abstract. Data refinement is a common approach to reasoning about programs, based on establishing that a concrete program indeed satisfies all the required properties imposed by an intended abstract pattern. Reasoning about programs in this setting becomes complex when use of pointers is assumed and, moreover, a well-known method for proving data refinement, namely the forward simulation method, becomes unsound in presence of pointers. The reason for unsoundness is the failure of the "lifting theorem" for simulations: that a simulation between abstract and concrete modules can be lifted to all client programs. The result is that simulation does not imply that a concrete can replace an abstract module in all contexts.

Our diagnosis of this problem is that unsoundness is due to interference from the client programs. Rather than blame a module for the unsoundness of lifting simulations, our analysis places the blame on the client programs which cause the interference: when interference is not present, soundness is recovered. Technically, we present a novel instrumented semantics which is capable of detecting interference between a module and its client. With use of special simulation relations, namely growing relations, and interpreting the simulation method using the instrumented semantics, we obtain a lifting theorem. We then show situations under which simulation does indeed imply refinement.

Keywords: Data Refinement, Separation Logic, Pointer Aliasing, Interference

1. Introduction

Data refinement is a well-established technique in reasoning about programs, used to connect an abstract design or program to a more concrete description [Hoa72]. This naturally reflects the way that many programs are conceived. Any book on data structures and algorithms, for instance, will describe algorithms working on mathematical objects, such as trees and graphs, without explicit mention of the computer's store. The

Correspondence and offprint requests to: ivanam@dcs.qmul.ac.uk

correctness of an implementation of the algorithm, the connection to its realization on a computer, involves data refinement.

This paper is about a problem that pointers cause in the theoretical foundations of data refinement. Stated simply, the usual (forward) simulation method for reasoning about refinement is unsound.

We stress that the problem concerns not the establishing of a relation between abstract and concrete – for which pointers cause no special foundational difficulties – but rather what we can infer from the existence of such relations. Given a simulation relation between concrete module M_{conc} and an abstract module M_{abs}, soundness should ensure that $C[M_{conc}]$ simulates $C[M_{abs}]$ for any program context $C[\cdot]$ for the modules. The problems arise when the client $C[\cdot]$ accesses module internals by following a pointer chain: it can then contradict the reasoning used originally to connect the modules.

In more technical terms, there are two facets of data refinement that we must consider, simulation and lifting [dRE98, HH90]. Simulation employs local relations connecting the states of abstract and concrete versions of a module, relations that do not talk about the states of clients or other modules. Simulation is a way to prove data refinement. Lifting then pushes the local relations up to the whole programming language, i.e., the relations are extended to include client program operations as well. If R is a relation showing that module M_{conc} simulates M_{abs}, and $C[\cdot]$ is a program context for the modules, then we want a result to the effect that there is an induced relation $C[R]$ showing that $C[M_{conc}]$ simulates $C[M_{abs}]$. (The lifting theorem is thus a close relative of the "abstraction theorem" or "main lemma of logical relations", which is a way to formalize data abstraction [Plo73, Rey83].)

In summary, the reason that pointers break soundness of data refinement (that simulation implies replaceability in all contexts) is that they break the lifting theorem for simulations.

The lifting theorem is usually left implicit in works on refinement, possibly because it is so obvious if one requires different modules to be refined using state described by disjoint variables. The problem with pointers is that this disjointness assumption is false. Pointers reveal that memory is a single giant array (the heap), so different modules use (in a sense) the same variable. Put another way, refinements of *different* modules must go into the *same* array.

Although it is often left implicit, the lifting theorem is essential to the effectiveness of simulation reasoning. For example, if a web server maintains a queue of incoming requests, it can be viewed as a client of a queue module. We can reason about the web server using an abstract model of the queue operations, provide a refinement of the queue by an array, and then the lifting theorem ensures that the reasoning we did about the client using the abstract model remains valid for the concrete model. It would be unfortunate if we had to re-verify the client with each refinement step, and that is what the lifting theorem saves us from.

The purpose of this paper is to explain this problem caused by pointers, and to develop a theory that offers at least a partial solution to the problem. Although they use the same giant memory-array, it is natural to ask for separate refinements to go to different portions of memory; otherwise, ostensibly separate modules, or a module and its client, can *interfere* with one another. It is the possibility of interference that causes problems for the lifting theorem. A complicating factor is that the portions of memory assigned to module and client, or to different modules, are not fixed once and for all, but rather change over time; the next section gives a particular example of this phenomenon, based on a memory manager module. It is this dynamic aspect of memory partitioning, rather than (only) the use of a single memory-array, that makes refinement in the presence of pointers subtle.

The work here employs ideas from separation logic, or rather its semantics, to describe this memory partitioning. Our account is done for an abstract storage model, certain partial commutative monoids, which means that it is not tied to any particular concrete model of memory.

We provide a full discussion of related work in Sections 3 and 13, but reproduce here an excerpt from one of the reviews of this paper, as it skilfully places the paper with respect to existing literature.

> "Previous work has addressed interference between clients and modules by focusing on types, static analyses, and fixed programming disciplines as means to prevent harmful interference between clients and modules. Furthermore, it made semantic assumptions suited to garbage-collected programs without pointer arithmetic, and it is encumbered by the syntactic overhead of object-oriented language features. By contrast, this paper encompasses data refinement for low level programs including pointer arithmetic. And it makes minimal assumptions that are directly motivated in semantic terms. Yet as the paper points out, the analysis here is quite pertinent to high level programs as well." *(From review of David Naumann.)*

In the next section of the paper we give an example that illustrates the problem tackled in this work,

and in the section after that we outline our way of reacting to the problem (by "blaming the client") and the new technical contributions (concerning healthiness conditions for command meanings and certain special kinds of simulation relations) that lead to its resolution. In Sections 4 and 5 we lay out the basics of the mathematical model. The syntax and semantics of the client programming language as well as the semantics of the modules are presented in Section 6. Section 7 gives an analysis of the healthiness conditions which ensure that commands access storage in circumscribed way; this is essential for our (abstract) account of interference. In Section 8, we introduce the main tools of our simulation method – growing coupling relations and trumping simulations – which are special simulations that both address technical difficulties and fit the conception of pointer programs. The following three sections form the technical core of our paper. In Section 9 we define the simulation method and prove the lifting theorem for an *instrumented semantics which detects interference*. In Section 10 we connect the instrumented semantics back to a standard semantics of programs, and explore the conditions under which the lifting theorem holds for standard semantics. In Section 11 we round up our results by stating the soundness of the forward simulation method with respect to refinement. This connection is needed because, as in [HHS86], we are distinguishing "simulation", which is about relations between concrete and abstract, from "refinement", which is about exchanging program fragments in larger contexts. Finally, we illustrate our theory with a detailed example in Section 12.

The example at the end of the paper suggests the "developmental" approach to verification of pointer programs that this work supports, where one starts from an abstract specification and successively refines to obtain a concrete implementation. This extends the modular, but not particularly "developmental", reasoning supported previously in separation logic. The main contribution of the work, though, is theoretical rather than one of methodology, and in the main body of the paper we use a small running example to illustrate the concepts and issues involved.

2. Illustration of the Problem

In Table 1 we give concrete and abstract descriptions of a toy memory manager, which provides procedures for allocating and deallocating certain binary cells corresponding to the `slink` struct definition. The abstract manager uses a finite set to represent the current collection of cells it holds. It doles these cells out nondeterministically through calls to talloc(), and receives them for recycling through calls to tfree(). In case the manager does not have enough cells, it asks the system for more through malloc(), analogous to how real memory managers sometimes ask for more memory from a system (e.g., Section 8.7 of [KR88]).

Our description of the abstract manager is informal. It uses the syntax of the C programming language, but it also uses operations on finite sets, which are not part of C language; the use of finite sets in the example should, we hope, be understandable to the reader. We also state the representation invariants for abstract and concrete modules in natural rather than mathematical language. Later, we will give a more precise rendering of the ideas expressed informally in this section.

In designing a concrete manager there are a number of choices concerning the operations and the data structure to be used. For example, sometimes a memory manager uses an acyclic free list, and sometimes cyclic lists are used; there are many choices in data structure arrangement which impact the performance of the manager. Further, there are different algorithms to search for, say, the best or first fit of an allocation request [Knu73]. In our illustration, we choose an acyclic free list, for simplicity. Since in our toy example the cells are all the same size we don't have to worry about fitness, and can just allocate any element. The resulting code for tfree() and talloc() is in the right column of Table 1.

To establish a connection between the concrete and abstract representations we can use a simulation relation, which we again state informally.

`Simulation Relation`: fl points to an acyclic linked list whose nodes are the elements of set S.

With this definition, we can then argue that the concrete implementations of tfree() and talloc() simulate (forward simulate) their abstract counterparts. Since this is standard we do not get into the details of arguing simulation between the two modules. We mention only that, for the argument to work, we must observe the preconditions stated (informally) with the tfree() procedures; we only consider calls to tfree() in which the preconditions hold. We return to this point about preconditions several times later.

We can now get to the heart of the problem alluded to in the Introduction. Consider the client program **User1** given below, which uses the operations in the memory manager module. The program allocates a new location, then dereferences it and deallocates it, after which ties a cycle of size one from x to x. If we execute

Table 1. Memory Manager Module

```
struct link{  //Shared Struct Definition
  struct link *nxt;
  int data; } slink;
```

Abstract Representation

```
FiniteSet S;

Rep Invariant :  S is a set of pointers to slink's

void tfree(slink *x) // Pre :  x ∉ S
  {S = S ∪ {x}; }


slink *talloc(){
  slink *res;
  if S = ∅ then
    res = (slink *)malloc(sizeof(slink));
  else
    pick I ∈ S; S = S \ {I}; res = I;
  return res; }
```

Concrete Representation

```
slink *fl;

Rep Invariant :  fl points to an acyclic linked list

void tfree(slink *x) // Pre :  x ∉  fl's list
     {slink *t;
      t = fl; fl = x; x → nxt = t; }

slink *talloc(){
     slink *res;
     if (fl = NULL)
          res = (slink *)malloc(sizeof(slink));
     else
          {res = fl; fl = fl → nxt; }
     return res; }
```

the client in conjunction with the concrete module, the free list gets corrupted. Namely, after the location is allocated, it is no longer a member of the free list, which is the internal representation of the module. However, after the location is freed it is put back on the free list, but client variable x still points to it; x and fl are aliases for the same location. When $\mathsf{x} \rightarrow \mathsf{nxt} = \mathsf{x}$ is executed, a cycle in the free list is created, and thus instead of the acyclic free list being maintained, a single-element cyclic one is produced. So, we have a situation where a client operation, $\mathsf{x} \rightarrow \mathsf{nxt} = \mathsf{x}$, alters the free list without using a module procedure. Even worse, this operation breaks the representation invariant of the concrete module, which requires the free list to be acyclic. This is a situation where pointers break abstraction: client code interferes with the internal representation of a module.

User1

```
void main(){
  slink *x;
  x = talloc();
  x → data = 5;
  tfree(x);
  x → nxt = x; }
```

The example's violation of abstraction can be turned into a problem for lifting simulations. Consider what happens when the client **User1** is executed together with the abstract module. In this case, after the first call to the tfree(x), location pointed to by x is returned to the set of available locations maintained by the manager. The operation $\mathsf{x} \rightarrow \mathsf{nxt} = \mathsf{x}$ will have no effect on the internal representation of the abstract module. But, when we consider concrete and abstract together, we contradict simulation. For, when the concrete $\mathsf{x} \rightarrow \mathsf{nxt} = \mathsf{x}$ ties a cycle in the free list, the simulation relation above cannot hold. From the perspective of data refinement, not only is the internal representation of the concrete module corrupted, but also the relation connecting it to the abstract internal representation is violated.

To summarize, we saw above that the concrete memory manager module simulates the abstract module, which certainly corresponds to intuition concerning the modules. But, we obtained that **User1** under the concrete interpretation does not simulate **User1** under the abstract interpretation of the module. This is the sense in which the lifting theorem breaks.

We can make one further step to connect to refinement. Suppose we only observe client states, we "ignore"

the internal states of the modules.[1] Then there is a client program where the concrete can do "more" than the abstract. We would expect, though, the abstract module to cover all of the behaviours of the concrete. This client program is a small extension of **User1**.

```
User2
void main(){
  slink *x, *y, *z;
  x = talloc();
  x → data = 5;
  tfree(x);
  x → nxt = x;
  y = talloc();
  z = talloc(); }
```

In this program, under the concrete interpretation, we will obtain that y and z are equal, whereas under the abstract interpretation they will not be.

The source of the problem in this example is *interference*: the statement $\mathsf{x} \rightarrow \mathsf{nxt} = \mathsf{x}$, which is not a module operation, tampers with the module's internal representation. What makes this kind of interference hard to deal with is that there are no *source code variables* in common between client and module. Rather, an indirectly named entity (a heap allocated pointer) is used as the conduit of interference. Put another way, we have *aliasing*, where the same heap cell is referenced, and accessed, by expressions with no variables in common.

3. Point of Departure

In our example we have used something that has been known for a long time: pointers can cause difficulties with data abstraction. The majority of work in reaction has taken place in the object-oriented types community (e.g., [Hog91, CNP01, BN05a]). Their reaction has been to outlaw aliasing between client and module, where a client obtains a pointer to an internal data structure used in a module. The approach has been, in the main, to place the blame on the module, by outlawing modules that leak pointers. This undeniably banishes certain instances of unintentional leakage. The problem, though, with this diagnosis is that, along with the bad modules, it tends to also outlaw very natural and important modules, such as the memory manager above.

Our point of departure in this work is to blame the client rather than the module. The diagnosis is that it is when the client *dereferences* module internals that we are in trouble as far as the lifting theorem goes; we aim for a situation where the lifting theorem works, when there is no interference. Our diagnosis does not prohibit aliasing between module and client. Rather, it allows the aliases to exist, and only prohibits them from being dereferenced at the wrong time [OYR09]. This is what allows our solution to apply to examples from systems programming, as well as more straightforward examples when a data structure is more explicitly encapsulated (such as when a linked list represents a queue, and no references are leaked).

Our approach admits many more modules than in the afore-mentioned work of the object-oriented types community, such as the memory manager above. **User1** above is an example of a bad client for the memory manager, but there are many good ones which hold aliases to the free list but do not wrongly use them. Essentially, any correct C program that uses free() will have such unharmful aliasing. We stress as well that this point is not limited to low-level C or assembly programs. For example, similar issues are raised by object-oriented programs that use idioms for managing resource pools (such as using thread pools in a web server).

In further support of our point of departure, one can find a non-surprising number of resources, such as hacker forums, articles, security books, etc., that bear witness to the problem of interference from bad clients. Some of them concern vulnerabilities obtained from misuse of one of the most widely used memory allocators [Lea01] (used in GNU C Library). They show techniques that use malicious clients for intentional corruption of the heap [Kae01, jp01, FOB05], often exploiting buffer overflow or double free() errors. On the other hand, the memory management module in question (dlmalloc) has a very good reputation:

[1] This "ignoring" the internal states will be done more formally in Section 11, with initialization and finalization operations for a module.

"This is not the fastest, most space-conserving, most portable, or most tunable malloc ever written. However it is among the fastest while also being among the most space-conserving, portable and tunable. Consistent balance across these factors results in a good general-purpose allocator for malloc-intensive programs." [Kae01]

We find this an excellent motive for putting the blame on clients instead of the module.

In the remainder of the paper we develop the mathematical technology which lets us "blame the clients", and rescue the lifting theorem. Our development makes use of ideas from separation logic to let us say when the client and module access separate resources. We are going to follow [HHS86, HH90] in the way we set up our theory. We will first set down an abstract model of states and of operations, we will then set down what counts as forward simulation and finally we will show the lifting theorem. Like in [HHS86, HH90] our account is oriented to partial correctness, in that it does not make guarantees about program termination.

Starting from the simple-sounding idea of using separation to rule out interference, it turns out to be (perhaps surprisingly) technically nontrivial to actually rescue the soundness of refinement. We mention the new technical contributions now, as a primer for the work that follows.

1. The semantics of commands in separation logic has principles – safety monotonicity and the frame property [YO02] – which describe a sense in which program operations use circumscribed resources; cirscumscription gives us a way to attack interference, by identifying situations when different program parts access separate resources. We find that that we need a new "healthiness condition" on command meanings, *contents independence*, to go along with safety monotonicity and the frame property. Every command needs some minimal heap (possibly empty) to run safely, and contents independence ensures that beyond this heap, the command does not care about the contents of the remaining locations of the heap in which the command was run.
2. One of the major obstacles to the soundness of the forward simulation method is that the simple allocation command does not simulate itself. In order to overcome this difficulty, we formulated a notion of *growing relations* to be used in the forward simulation method. These relations, that require the abstract state to be "smaller" than the concrete state, somewhat restrict the method, but still push its limits to work well in presence of pointers and still cover a good range of examples. Apart from that, the idea of growing relations also coincides with the view that refinement starts with an abstract data structure that does not mention store and ends up with some data structure representable in the heap.
3. We use of certain special simulation relations that we call *trumping simulations*, which say that when the abstract program commits an error (akin to a memory fault), then simulation holds automatically (so that faulting trumps). Put another way, it is the responsibility of the designer of the abstract program, and not the designer of the programming language, to ensure that certain faults do not occur.

It is exactly these things – the healthiness properties of commands (safety monotonicity, the frame porperty and contents independence), together with the requirement that only growing relations are to be used with trumped simulations method – that allow for the soundenss of the forward simulation method.

(This section has not discussed research that has taken place since the preliminary version [MTSO04] of this work appeared, subsequent work that appears to take the same point of departure but utilizing different machinery; that and other related work is discussed in Section 13.)

4. The Storage Model

In traditional works on data refinement each data type or module is associated with a set of values, and partitioning between different modules, or between client and module, is achieved using cartesian products. For example, if a queue module has state set Q, and its client has state set Z, then the combined system has state set $Q \times Z$.

While static partitioning of this kind is very useful when applicable, it rules out useful programming idioms. As we saw in our memory manager example, the partitioning between module and client is not set down once and for all, but changes over time; cells move back and forth between the manager and the client. In this section we define a mathematical structure of state spaces that allows us to model the dynamic partitioning that arises naturally when thinking about pointer programs. Our structure is that of a set equipped with a partial binary composition operator. We think of the elements of the set as describing *portions of state* rather than entire global states, and an equation $h = h_0 \cdot h_1$ represents a partitioning of

a heap h into "heaplets" h_0 and h_1. In the general model we will treat our state spaces axiomatically, and then we will give a particular concrete model based on the classic random access memory (RAM) model.

The General Model. Our general model is on the level of the abstract model theory of BI [POY04], rather than the single model used in separation logic [IO01, Rey02]. We assume that we are given two disjoint countably infinite sets of variables – client variables Var_c and module variables Var_m, and a set Val of values. Let S_c denote the set of all functions from client variables to values, and similarly let S_m denote a set of all maps from module variables to values. Then the set of all stacks S is defined to be the set of all combinations of module and client stacks.

$$S_c = \mathsf{Var}_c \to \mathsf{Val} \qquad S_m = \mathsf{Var}_m \to \mathsf{Val}$$
$$S = \{s_c \cup s_m \mid s_c \in S_c \ \wedge \ s_m \in S_m\}.$$

We use metavariable s to range over elements of S.

Because we are concerned with pointers, the state has a second component, the heap. Abstractly, we assume that H (for *heaps*) is a set equipped with a structure of a cancellative partial commutative monoid $(H, \cdot, e)$. That is, $\cdot : H \times H \rightharpoonup H$ is a partial function that is commutative and associative, and $e \in H$ functions as a unit for all elements of H.[2] The cancellativity property means that, for each h, the partial function $h \cdot - : H \rightharpoonup H$ is injective. The cancellativeness of the $\cdot$ is an important requirement and the consequences that it has for our theory will be explained later on.

The subheap order $\sqsubseteq$ is induced by $\cdot$ in the following way

$$h_1 \sqsubseteq h_2 \iff \exists h_3.\ h_1 \cdot h_3 \downarrow \ \wedge \ h_2 = h_1 \cdot h_3.$$

Here $\downarrow$ denotes the fact that the composition $h_1 \cdot h_3$ is defined. Two heaps h_1 and h_2 are called disjoint, denoted $h_1 \# h_2$, if $h_1 \cdot h_2$ is defined.

The RAM Model. The Random Access Memory (RAM) model is an instance of the general storage model. Let H be the set of finite partial functions from a given set of addresses Ptr to the set of values Val. The set of addresses is a subset of the set of positive integers; this allows address arithmetic.

$$H = \mathsf{Ptr} \rightharpoonup_{fin} \mathsf{Val}, \text{ where } \mathsf{Ptr} = \{1, 2, \ldots\} \quad \text{and} \quad \mathsf{Val} \subseteq \{\ldots, -1, 0, 1, \ldots\}.$$

We now define the operation $\cdot$, making sure that all the requirements of the general storage model are respected. We say that two heaps are disjoint $h \# h'$ if their domains are disjoint, i.e. $\mathsf{dom}(h) \cap \mathsf{dom}(h') = \emptyset$. The combination $h \cdot h'$ of two heaps is defined only when they have disjoint domains

$$h \cdot h'(a) = \begin{cases} h(a), & a \in \mathsf{dom}(h), \\ h'(a), & a \in \mathsf{dom}(h'), \\ \text{undefined}, & \text{otherwise.} \end{cases}$$

When $h \# h'$ fails, we stipulate that $h \cdot h'$ is undefined.

It can be easily seen that the RAM model is a cancellative partial commutative monoid.

Note. For proving our formal results, we will assume that the underlying storage model is the general one. Then all the results also apply to the RAM model, and since it is much closer to intuition, for most of our examples we will assume it as the underlying storage model.

In the general model (and hence also in the RAM model), we denote by Σ the set of all *states*, i.e., the set of all pairs of stacks and heaps $\Sigma = S \times H$. So, elements of Σ are pairs (s, h) which consist of the stack and the heap component. If we want to be explicit about the module and client parts of the stack, we use s_c or s_m instead of s.

5. Separation Logic and Specification Statements

The models in the previous section use some of the mathematical structure lying behind separation logic. It would be theoretically possible, in most of what follows, to ignore the logic completely, to work purely at

[2] Unity, associativity and commutative laws for a partial function are interpreted to mean that both sides are either undefined, or they are both defined and equal.

a semantic level. But, it will be convenient, particularly in examples, to call upon the logic to give concise specifications of module operations, using an analogue Schwarz's notion of generic command [Sch77], the largest relation determined by a pre/post pair. A related notion has played a prominent role in work on program refinement [Bac78, Bac80], where it is now called a "specification statement" [MRG88], and we will adopt this as our terminology. (The reader who is already familiar with separation logic can safely skip forward to the next section, and refer back as necessary.)

Separation Logic. Separation logic [IO01, Rey02] uses a model of Logic of Bunched Implications [OP99], designed with an assumption that the RAM is the underlying storage model.

$$\begin{array}{lll} P,Q,R ::= & B \mid E_1 \mapsto E_2 & \text{Atomic formulae} \\ & \mid \mathsf{false} \mid P \Rightarrow Q \mid \exists x.P & \text{Classical Logic} \\ & \mid \mathsf{emp} \mid P * Q \mid \mathsf{true} & \text{Spatial assertions.} \end{array}$$

Assuming

$$[\![E]\!]_s \in \mathsf{Int} \qquad [\![B]\!]_s \in \{\mathsf{true}, \mathsf{false}\},$$

where

$$[\![-]\!]_s : S \to \mathsf{Val}, \qquad \mathsf{Val} = \{\ldots, -1, 0, 1, \ldots\},$$

the semantics of the assertion language is given bellow.

$$\begin{array}{lll} (s,h) \models B & \text{iff} & [\![B]\!]_s = \mathsf{true} \\ (s,h) \models E \mapsto F & \text{iff} & \{[\![E]\!]_s\} = \mathsf{dom}(h) \text{ and } h([\![E]\!]_s) = [\![F]\!]_s \\ (s,h) \models \mathsf{false} & & \text{never} \\ (s,h) \models P \Rightarrow Q & \text{iff} & \text{if } (s,h) \models P \text{ then } (s,h) \models Q \\ (s,h) \models \exists x.P & \text{iff} & \exists v \in \mathsf{Int}.\ (s[x \mapsto v], h) \models P \\ (s,h) \models \mathsf{emp} & \text{iff} & h = e \text{ is the empty heap} \\ (s,h) \models P * Q & \text{iff} & \exists h_0, h_1.\ h_0 \# h_1,\ h = h_0 \cdot h_1,\ (s,h_0) \models P \text{ and } (s,h_1) \models Q \end{array}$$

Here, $f[m \mapsto n]$ denotes the same function as f, except that it assigns value n to entry m.

The interpretation of assertion $E \mapsto F$ asserts that the heap component of the state contains exactly one location with address E and the contents of that location is F. We write $E \mapsto -$ as shorthand for $\exists y.\, E \mapsto y$ where y is not free in E, and also $E \mapsto E_1, E_2, \ldots, E_n$ as shorthand for $E \mapsto E_1 * E{+}1 \mapsto E_2 * \ldots * E{+}n{-}1 \mapsto E_n$. The assertion emp says that the heap component of the state is empty, while the separating conjunction $P * Q$ asserts that the state can be split into two disjoint parts, such that in one part P holds and in the other Q holds. Other assertions have a standard interpretation.

We define a predicate $\mathsf{list}(\alpha, x)$, where α ranges over sequences of integers and x over integers. The predicate $\mathsf{list}(\alpha, x)$ is defined inductively on the sequence α by

$$\mathsf{list}(\epsilon, x) \stackrel{def}{=} x = nil \wedge \mathsf{emp}, \qquad \mathsf{list}(a\alpha, x) \stackrel{def}{=} x = a \wedge \exists y, v.\ x \mapsto y, v * \mathsf{list}(\alpha, y)$$

where ϵ represents the empty sequence and *nil* is the integer -1. Here, v records the value stored in the list, and such values are not important for the purpose of this example. This predicate says that x points to a non-circular singly-linked list whose addresses are the elements of the sequence α (this is called a "Bornat list" in [Rey02]).

These definitions were for the RAM model. In the abstract storage model, we specify properties of the heap using subsets of Σ directly, instead of syntactic formulas. We call such subsets of Σ *predicates*, and in the theory use the semantic version of separating conjunction:

$$p, q \in \mathsf{Pred} \stackrel{def}{=} \wp(\Sigma) \qquad \mathsf{true} \stackrel{def}{=} \Sigma$$
$$p * q \stackrel{def}{=} \{(s, h_0 \cdot h_1) \mid h_0 \# h_1 \ \wedge\ (s,h_0) \in p \ \wedge\ (s,h_1) \in q\}.$$

Specification Statements. If p and q are predicates, and X is a set of variables, then we define the relation

$$\{p\} - \{q\}[X] \quad \subseteq \quad \Sigma \times (\Sigma \cup \{\mathsf{wrong}\})\ .$$

The special state wrong signifies a memory fault such as a dangling ot null pointer dereference, and we assume it does not belong to the set of states Σ. It is the greatest relation satisfying the Hoare triple $\{p\} - \{q\}$,

which modifies only variables in X, and which in addition satisfies the "frame rule" of separation logic (to be layed out later in Section 7). An explicit characterization of this relation is given by the formulas

$$\begin{array}{l}(s,h)\big[\{p\}-\{q\}[X]\big]\mathsf{wrong}\\ \stackrel{def}{\iff}\ (s,h)\notin p * \mathsf{true}\end{array}$$

$$\begin{array}{l}(s,h)\big[\{p\}-\{q\}[X]\big](s',h')\\ \stackrel{def}{\iff}\ (1)\ s(y)=s'(y)\ \text{for all variables}\ y\notin X\ \text{and}\\ \qquad (2)\ \forall h_0,h_1.\ \ h_0\cdot h_1=h\ \wedge\ (s,h_0)\in p\\ \qquad\qquad \Longrightarrow \exists h_0'.\ h_0'\#h_1\ \wedge\ h_0'\cdot h_1=h'\ \wedge\ (s',h_0')\in q.\end{array}$$

The first equivalence ensures that $\{p\}-\{q\}[X]$ will run safely in a state (s,h) just when predicate p holds in some substate (s,h_p) of (s,h). The second equivalence defines how the greatest relation changes the state. The condition (1) makes sure that $\{p\}-\{q\}[X]$ can modify only variables listed in X. Condition (2) defines $\{p\}-\{q\}[X]$ to demonically dispose of the initial heap h_0 which satisfies p and then, to angelically choose a heap h_0' from q and combine it with the remaining initial heap h_1 in order to get the final heap h'. We refer to [OYR09] for more information on this greatest relation.

Using specification statements we can describe the semantics of primitive client commands in the RAM model, following on from the "small axioms" of [ORY01]. For each primitive command $\mathsf{cl\_op}$, we define the relation $[\![\mathsf{cl\_op}]\!]$ as follows.

$$\begin{array}{lcl}[\![[E]:=F]\!] & = & \{E\mapsto -\}-\{E\mapsto F\}[]\\ [\![\mathsf{free}(E)]\!] & = & \{E\mapsto -\}-\{\mathsf{emp}\}[]\\ [\![x:=\mathsf{alloc}()]\!] & = & \{\mathsf{emp}\}-\{x\mapsto -\}[x]\\ [\![x:=E]\!] & = & \{x=n\wedge\mathsf{emp}\}-\{x=E[n/x]\wedge\mathsf{emp}\}[x]\\ [\![x:=[E]]\!] & = & \{x=m\ \wedge\ E\mapsto n\}-\{x=n\ \wedge\ E[m/x]\mapsto n\}[x]\end{array}$$

where x,m,n are distinct variables and where m and n do not appear in E.

Note that the small axiom for $\mathsf{alloc}()$ is oriented to the idealized assumption of the RAM model, that there be an infinite supply of addresses (and so always a new one to choose). To model a situation where $\mathsf{alloc}()$ could fail, in a model with bounded memory, we could add a disjunct to the postcondition – say, $\cdots\vee(x=-1\wedge\mathsf{emp})$ – to cover the possibility of failure of the allocator. A disjunctive postcondtion of this sort is in fact used in several program analysis tools for the C language based on separation logic.

Remark on Ghost Variables. To instantiate a specification statement, one usually needs to know which variables are not free in a command (for example, to apply the structural rule of substitution for Hoare triples [Rey02]). Alternatively, one can identify a concept of "ghost variable", variables that do not appear in any programs. For example from the small axiom for lookup

$$[\![x:=[E]]\!]\quad=\quad\{x=m\ \wedge\ E\mapsto n\}-\{x=n\ \wedge\ E[m/x]\mapsto n\}[x]$$

we would like to be able to instantiate n with (say) the number 7, and we cannot do that if the program reads from n. The problem is that our formalism has not identified either a concept of free or ghost variable.

We can, though, explain how to understand the examples (like the small axiom) without complicating our theory with concepts of free or ghost variables. Generally, if we write

$$\{P(n)\}-\{Q(n)\}$$

as a specification statement where we would like to think of n as a ghost variable, then we regard it as shorthand for the explicit meet

$$\bigsqcap_{n\in\mathtt{Val}}\{P(n)\}-\{Q(n)\}$$

where $\mathtt{Val}$ is the set of values in the storage model in question and $\sqcap$ is the meet of relations satisfying some healthiness conditions in Section 7 – this meet exists as shown in Lemma 1. Here, $P(-)$ and $Q(-)$ are regarded as functions from values to predicates, and we are essentially using the semantics of universal quantification in the metalanguage of ordinary (rigorous but informal) mathematics to play the role of interpreting ghost variables using universal quantification outside of Hoare triples. The variable n is in the metalanguage, not an object language variable, and hence there is no read-dependence on it.

So, the small axiom for heap lookup would be more officially rendered as

$$[\![x := [E]]\!] \quad = \quad \bigsqcap_{n,m \in \mathtt{Val}} \{x = m \ \wedge \ E \mapsto n\} - \{x = n \ \wedge \ E[m/x] \mapsto n\}[x]$$

where we consider n and m as metavariables in mathematics ranging over values, rather than as program variables in the syntax of the programming language.

Similarly, existential quantification involving a ghost variable can be understood using the join of predicates, which becomes the set union in this case. For example, the heap predicate $\exists\alpha.\ \mathsf{list}(\alpha, fl)$ involving a variable α ranging over sequences in Table 5, can be understood as its semantics $\sqcup_\alpha\ \mathsf{list}(\alpha, fl)$, which has no free variables other than program variables.

With this being understood, we will continue to use Hoare triple form in the examples, without the explicit meet or join. In the running examples, α, α', n, m and a will be ghost variables, and we expect that the reader can translate to the $\sqcap$- or $\sqcup$-form if need be.

Finally, we remark that this issue concerning ghost variables has only to do with our examples rather than our technical results.

Precise Predicates and Representation Invariants. Finally, in the development of our theory we will want a way to talk about when a client program interferes with the internal state of a module. So, we need a way to describe the internal state of a module, when not all of the cells are directly named by program variables. For this we will employ a special kind of predicate defined on states – *precise predicates*. A precise predicate is a unary relation that unambiguously determines a portion of a possibly larger piece of state [OYR09, Rey05].

Definition 1 (Precise predicate). A predicate $p \subseteq \Sigma$ is *precise* if for any state (s, h) there is at most one subheap $h_0 \sqsubseteq h$, such that $(s, h_0) \in p$.

In brief, the internal state of a module will be described by a precise predicate (its "representation invariant").

Some informal remarks on precise predicates are in order. We have often observed a misconception, that the use of precise predicates is very restrictive in practice. *However, we emphasize that a data structure almost always has a natural associated precise predicate.* Indeed, precise predicates formalize an idea of Richard Bornat's, where data structures are described by predicates corresponding to traversal procedures [Bor00]. It is in a way easier to describe the idea in terms of deletion rather than traversal. If you know what the shape of a data structure is, you can write a deterministic "deletion procedure" that frees all its elements. Suppose you have the deletion procedure: from this we can define a precise predicate p. We assume that the deletion procedure removes, but does not otherwise mutate any cells or mutate any variables. For a state (s, h), if running the deletion procedure faults (because it encounters a dangling pointer) then we say $(s, h) \notin p$. Otherwise, consider the heap h'' obtained on successful termination of the deletion procedure run on h, and form the heap $h' = h - h''$. The heap h' consists of those cells that were removed by the deletion procedure, and it is the unique subheap of h such that $(s, h'') \in p$.

Thus, if you can define a deletion procedure for your data structure, then there is a natural corresponding precise predicate. Further, if you were not able to write such a procedure, you would hardly be in a position to write a predicate describing the data structure. In this sense, we claim that the restriction to precise predicates (or to predicates of the form $p * \mathtt{true}$, for p precise), is a natural rather than drastic restriction.

Examples. The notion of precise predicate makes sense in the abstract storage models; we illustrate it with examples drawn from the RAM model. Examples of precise predicates are emp, $x \mapsto -$, $\mathsf{list}(\alpha, x)$, and $P * Q$ when both P and Q are precise. Another example is the predicate $\mathsf{Set}(\alpha, S)$ where, similarly as in list predicate,

$$\mathsf{Set}(\epsilon, S) \stackrel{def}{=} S = \emptyset \ \wedge \ \mathsf{emp}, \qquad \mathsf{Set}(a \cdot \alpha, S) \stackrel{def}{=} a \in S \ \wedge \ a \mapsto -, - * \mathsf{Set}(\alpha, S - \{a\})$$

where $S - \{a\}$ is set difference.

In the toy memory manager, $I = \exists\alpha.\, \mathsf{list}(\alpha, x)$ is a rigorous description of the representation invariant given informally in Table 1. When a client program is interacting with the manager we can use an assertion of the form $I * (\mathit{client\ state})$ to give a partial description of the combined state of the client and the manager. Before a $\mathsf{tfree}(y)$ operation we might have $I * y \mapsto - * x \mapsto -$, and then afterwards we would have $I * x \mapsto -$.

Table 2. Language of Client Programs, uninstantiated

$$
\begin{array}{rcll}
E & ::= & var \mid \ldots & [\![E]\!]_s \in \mathsf{Val}, var \in \mathsf{Var}_c, \\
B & ::= & \mathsf{false} \mid B \Rightarrow B \mid E = F \mid \ldots & [\![B]\!]_s \in \{\mathsf{true}, \mathsf{false}\} \\
c_{user} & ::= & \mathsf{cl\_op} & \\
 & & \mid \mathsf{mod\_op} & \\
 & & \mid c_{user}; c_{user} & \\
 & & \mid \textbf{if } B \textbf{ then } c_{user} \textbf{ else } c_{user} & \\
 & & \mid \textbf{while } B \textbf{ do } c_{users}, &
\end{array}
$$

$\mathsf{cl\_op} \in \mathsf{COp}$, $\mathsf{mod\_op} \in \mathsf{MOp}$,
$\mathsf{COp}, \mathsf{MOp}$ – sets of primitive client and module commands.

Here, the y cell has "moved" into I. This illustrates how using $*$ allows us to track dynamic partitioning between the module and client states, where the cells belonging to each change over time.

Finally, an example of an imprecise predicate is $(x \mapsto -) \vee (y \mapsto -)$. This says that either x or y is allocated, but not which. It is an example of a predicate that does not pick a unique subheap of a larger heap: it would identify either the $(x \mapsto -)$ or the $(y \mapsto -)$ subheap from the larger heap $(x \mapsto -) * (y \mapsto -)$. It is the kind of predicate that we will not want to use as the representation invariant of a module.

In general, precise predicates are closed under $*$ and $\wedge$, but not $\vee$. They are, however, closed under special disjunctions corresponding to if-then-else assertions – $(B \wedge P) \vee (\neg B \wedge Q)$ where P and Q are precice and B is heap-independent – which require the disjuncts to be disjoint.

Taking into account that $\cdot$ is cancellative, if p is a precise predicate, then it induces a unique splitting of the state. The uniqueness of the splitting induced by a precise predicate is particularly important, since the nondeterminism of $*$ interacts badly with modularity, as already demonstrated in work with separation logic [OYR09, O'H07, Bro07].

Notation. For a precise predicate $p \subseteq \Sigma$ and a state $(s, h) \in p$, we denote by h_p the unique subheap of a heap h such that $(s, h_p) \in p$. (To be completely consistent we should write $h_{p,s}$ to be explicit about the dependence of this subheap on s, but s will always be clear from context and no confusion is likely to arise in the use of this notation.) We point out here that such a subheap of heap h does not always exist and consequently, h_p is not always defined. However, we consider specific states $(s, h) \in p$ (and not the arbitrary ones) and use this notation only when such a heap exists.

6. The programming language

6.1. Syntax

We will use a simple programming language that distinguishes two kinds of primitive operations: the client operations and the module operations. The syntax of the language is given in Table 2. It is a version of the usual language of while programs, parametrized by a finite set COp of primitive client operations and a finite set MOp of module operations, and allowing for additional boolean expressions other than those specified. It is the language in which client programs are written, giving us the "contexts" for a module, that we will need for stating our lifting theorem. Since the language is for client programs, its expressions E, boolean expressions B and module operations MOp will not access module variables.

We do not specify all the details of the language. For instance, we just say that expressions take values from the set Val, but we do not define what Val is in the abstract language. When the concrete implementation of the abstract language is decided, all the domains and the commands are appropriately instantiated.

Example. We can instantiate the client language as in Table 3. The set Val is instantiated to be a subset of the integer values, and the primitive client operations can be set to be the standard commands for manipulating the state. Here, $x := E$ is standard assignment, command $[E_1] := E_2$ denotes a heap update, command $x := [E]$ denotes a heap lookup, and $\mathsf{alloc}()$ and $\mathsf{free}()$ are built-in operations for allocating and disposing memory, respectively. Note that this instantiation allows address arithmetic. We assume that the language expressions (B, E, F) do not access heap storage, i.e. they depend only upon the stack component of the state. In our examples we will use the concrete client language given in Table 3. Also, an instance of

Table 3. Client Language for RAM Example

$$
\begin{array}{rcl}
E, F & ::= & \ldots \mid int \mid E + F \mid E \times F \mid E - F, \quad int \in \mathsf{Int}, \\
B & ::= & \ldots \mid E \leq F \\
\mathsf{cl\_op} & ::= & x := E \quad x \in \mathsf{Var}_c \\
 & & \mid [E_1] := E_2 \\
 & & \mid x := [E] \\
 & & \mid \mathsf{alloc}() \\
 & & \mid \mathsf{free}(E) \\
x \in \mathsf{Var}_c & & \mathsf{Int} = \{\ldots -1, 0, 1, \ldots\},\ \mathsf{Val} \subseteq \mathsf{Int}
\end{array}
$$

Table 4. Module Operations for Toy Memory Manager

$$
\begin{array}{rcl}
\mathsf{mod\_op} & ::= & \mathsf{talloc}() \\
 & & \mid \mathsf{tfree}()
\end{array}
$$

a module language is given in Table 4. In fact, from here on, we will use either of the two implementations (abstract and concrete) of the toy memory manager module, given in Table 1 as a running example.

6.2. Semantics of Modules

A command of the programming language will denote a binary relation

$$r \subseteq \Sigma \times (\Sigma \cup \{\mathsf{wrong}\})$$

between states and states with an additional state, wrong. The special state wrong is produced when a command dereferences a location that is not allocated in the current state.

A module comprises a representation invariant and a collection of operations that preserve the invariant.

Definition 2 (Module). A (semantic) module is a pair (p, η), where $p \subseteq \Sigma$ is a precise predicate (the invariant) and $\eta : \mathsf{MOp} \longrightarrow \mathcal{P}(\Sigma \times (\Sigma \cup \{\mathsf{wrong}\}))$ is a function (the operation assignment) which assigns a command meaning to each module operation. We require that, for all $\mathsf{mod\_op} \in \mathsf{MOp}$ and pairs of states $(s, h), (s', h')$,

$$
\begin{array}{l}
(\neg(s,h)[\eta(\mathsf{mod\_op})]\mathsf{wrong} \ \wedge \ (s,h) \in p * \mathsf{true} \ \wedge \ (s,h)[\eta(\mathsf{mod\_op})](s',h')) \\
\quad \Longrightarrow \quad (s',h') \in p * \mathsf{true}.
\end{array}
$$

The preservation property in this definition is perhaps not the first one that comes to mind. It says that the invariant must be preserved only when the operation does not go wrong. This stems from our way of avoiding explicit preconditions in our theory, replacing them by the use of relations determined by specification statements, a point illustrated in the following example.

Our definition of module does not include operations for initialization, which would establish the invariant, or finalization, which would delete its storage. In developing our theory of simulation it is simpler to do without them. Their inclusion is, though, conceptually straightforward, and we will make further remarks on them in Section 11.

Example. Concrete and abstract versions of the memory manager module are given in Table 5. Where two specification statements are associated with an operation we mean that it denotes the intersection of the corresponding relations. These are theoretically rigorous versions of the informal example given in Table 1.

We can now make further remarks on the connection between wrong and preconditions. Our intention in the informal presentation in Table 1 was that the invariant had to be preserved only when the precondition was satisfied. Consider the specification statements for tfree() in Table 5. They are such that if x is not allocated in the pre-state then the command can do anything, including delivering wrong. This then meshes with the preservation property in Definition 2 to give the desired effect.

In making these remarks we stress that, while preconditions are used in specification statements as a convenient way to define relations in examples, preconditions are not themselves part of our theory.

Table 5. Toy Memory Manager Module Specification

Concrete toy memory manager module:	$(\exists\alpha.\ \mathsf{list}(\alpha, fl), \eta_c)$, where
$\eta_c(\mathsf{talloc}())\ =$	$\{\mathsf{list}(a\cdot\alpha, fl)\} - \{\mathsf{list}(\alpha, fl) * a \mapsto -,- \ \wedge\ x = a\}[x, fl]$
	$\{\mathsf{list}(\epsilon, fl)\} - \{\mathsf{list}(\epsilon, fl) * x \mapsto -,-\}[x, fl]$
$\eta_c(\mathsf{tfree}())\ =$	$\{\mathsf{list}(\alpha, fl) * x \mapsto -,-\} - \{\mathsf{list}(x\cdot\alpha, fl)\}[fl]$
Abstract toy memory manager module:	$(\exists\alpha.\ \mathsf{Set}(\alpha, S), \eta_a)$, where
$\eta_a(\mathsf{talloc}())\ =$	$\{\mathsf{Set}(a\cdot\alpha, S)\} - \{\mathsf{Set}(\alpha, S) * a \mapsto -,- \ \wedge\ x = a\}[x, S]$
	$\{\mathsf{Set}(\epsilon, S)\} - \{\mathsf{Set}(\epsilon, S) * x \mapsto -,-\}[x, S]$
$\eta_a(\mathsf{tfree}())\ =$	$\{\mathsf{Set}(\alpha, S) * x \mapsto -,-\} - \{\mathsf{Set}(x\cdot\alpha, S)\}[S]$.

6.3. Semantics of Commands

For our general semantics we assume a module semantics (p, η) as in Definition 2, and we additionally assume that, for each primitive client command $\mathsf{cl\_op}$, a relation

$$[\![\mathsf{cl\_op}]\!] \subseteq \Sigma \times (\Sigma \cup \{\mathsf{wrong}\})$$

is given, which satisfies the conditions below:

1. $[\![\mathsf{cl\_op}]\!]$ does not write to module variables:
$$\forall (s_m \cup s_c, h), (s'_m \cup s'_c, h') \in \Sigma.\ \ (s_m \cup s_c, h)[\![\mathsf{cl\_op}]\!](s'_m \cup s'_c, h') \Longrightarrow s_m = s'_m$$
2. $[\![\mathsf{cl\_op}]\!]$ does not read from module variables:
$$\begin{array}{l}\forall (s_m \cup s_c, h), (s'_m \cup s'_c, h') \in \Sigma,\ s''_m \in S_m.\\ \quad ((s_m \cup s_c, h)[\![\mathsf{cl\_op}]\!](s'_m \cup s'_c, h') \Longrightarrow (s''_m \cup s_c, h)[\![\mathsf{cl\_op}]\!](s''_m \cup s'_c, h')) \ \wedge\\ \quad ((s_m \cup s_c, h)[\![\mathsf{cl\_op}]\!]\mathsf{wrong} \Longrightarrow (s''_m \cup s_c, h)[\![\mathsf{cl\_op}]\!]\mathsf{wrong}).\end{array}$$

In the case of the RAM/Memory Manager instantiation of our general language, two module models were given in Table 5 and $[\![\mathsf{cl\_op}]\!]$ for each client command was given by specification statements in Section 5. The relations for the primitive operations are also required to satisfy certain *healthiness conditions* which we set out in the next section.

The relation $\rightsquigarrow_{(p,\eta)} \subseteq (c_{user} \times \Sigma) \times (\Sigma \cup \{\mathsf{wrong}\})$ is the standard operational semantics and it is given in Table 6. It is a "big step" semantics. In the table, we use the semantics $[\![B]\!] : S \to \{\mathsf{true}, \mathsf{false}\}$ of boolean expressions, which satisfies the below condition that B does not read from module variables:

$$\forall s_m, s'_m \in S_m.\, \forall s_c \in S_c.\ \ [\![B]\!]_{(s_m \cup s_c)} = [\![B]\!]_{(s'_m \cup s_c)}.$$

We give a second semantics that detects interference. The "instrumented semantics" is a relation $\rightsquigarrow^i_{(p,\eta)} \subseteq (c_{user} \times \Sigma) \times (\Sigma \cup \{\mathsf{wrong}, \mathsf{av}\})$. Here, av indicates an *access violation*. A client program communicates with a module through the provided module operations and this is the only accepted manner in which they can exchange information. An access violation is encountered when a client program improperly reads from or writes to a module's part of the state.

The instrumented semantics has all the rules given in Table 6, with the alteration that $K \in \Sigma \cup \{\mathsf{wrong}\} \cup \{\mathsf{av}\}$ can take on the value av. In addition to these there is one more rule, which detects interference:

$$\frac{\neg(s,h)[\![\mathsf{cl\_op}]\!]\mathsf{wrong} \qquad (s,h) \models p * \mathsf{true} \qquad h = h_p \cdot h_u \qquad (s,h_u)[\![\mathsf{cl\_op}]\!]\mathsf{wrong}}{\mathsf{cl\_op}, (s,h) \rightsquigarrow^i_{(p,\eta)} \mathsf{av}}\ .$$

State $(s, h_p) \in p$ in this rule is the substate of (s, h) uniquely determined by predicate p and (s, h_u) denotes the rest of the state. The uniqueness of this splitting follows from the fact that p is precise and this enables us to determine which portion of the state belongs to whom.

The idea behind this rule is as follows. Suppose that a client operation does not go wrong, but does access module state. Then, when we subtract the module state and run the client operation again, it will go wrong, because it attempts to access the state which has been removed. The decomposition $h = h_p \cdot h_u$ is how we formally model this intuition of subtracting the state.

Table 6. The language semantics

$$\frac{(s,h)[[\![\mathsf{cl\_op}]\!]](s',h')}{\mathsf{cl\_op},(s,h) \rightsquigarrow_{(p,\eta)} (s',h')} \qquad \frac{(s,h)[\eta(\mathsf{mod\_op})](s',h')}{\mathsf{mod\_op},(s,h) \rightsquigarrow_{(p,\eta)} (s',h')} \qquad \frac{c_1,(s,h) \rightsquigarrow_{(p,\eta)} (s',h') \;\; c_2,(s',h') \rightsquigarrow_{(p,\eta)} K}{c_1;c_2,(s,h) \rightsquigarrow_{(p,\eta)} K}$$

$$\frac{(s,h)[[\![\mathsf{cl\_op}]\!]]\mathsf{wrong}}{\mathsf{cl\_op},(s,h) \rightsquigarrow_{(p,\eta)} \mathsf{wrong}} \qquad \frac{(s,h)[\eta(\mathsf{mod\_op})]\mathsf{wrong}}{\mathsf{mod\_op},(s,h) \rightsquigarrow_{(p,\eta)} \mathsf{wrong}} \qquad \frac{[\![B]\!]_s = \mathsf{true} \;\; c_1,(s,h) \rightsquigarrow_{(p,\eta)} K}{\textbf{if } B \textbf{ then } c_1 \textbf{ else } c_2,(s,h) \rightsquigarrow_{(p,\eta)} K}$$

$$\frac{c_1,(s,h) \rightsquigarrow_{(p,\eta)} \mathsf{wrong}}{c_1;c_2,(s,h) \rightsquigarrow_{(p,\eta)} \mathsf{wrong}} \qquad \frac{[\![B]\!]_s = \mathsf{false} \;\; c_2,(s,h) \rightsquigarrow_{(p,\eta)} K}{\textbf{if } B \textbf{ then } c_1 \textbf{ else } c_2,(s,h) \rightsquigarrow_{(p,\eta)} K}$$

$$\frac{[\![B]\!]_s = \mathsf{true} \;\; c;\textbf{while } B \textbf{ do } c,(s,h) \rightsquigarrow_{(p,\eta)} K}{\textbf{while } B \textbf{ do } c,(s,h) \rightsquigarrow_{(p,\eta)} K} \qquad \frac{[\![B]\!]_s = \mathsf{false}}{\textbf{while } B \textbf{ do } c,(s,h) \rightsquigarrow_{(p,\eta)} (s,h)}$$

$K \in \Sigma \cup \{\mathsf{wrong}\}, \qquad \mathsf{mod\_op} \in \mathsf{MOp}$

As the reader will have noticed, there are two faulty states, namely, av and wrong. Intuitively, wrong is used when to indicate dereferencing of unallocated memory, while av identifies an illegal dereference of allocated memory. Here, "illegal" means an access by the client to memory owned by the module. More crucially, we found a technical need for this distinction: if we had only one faulty state, the safety monotonicity property considered in the next section would fail.

Example. To illustrate how the new rule works, consider a situation where a client program consisting of a statement $x{\rightarrow}nxt{=}x$ is interacting with concrete toy memory manager module given in Table 5. Suppose that the current global state is the following state:

$$(s,h) = ([fl \mapsto 42, x \mapsto 42, y \mapsto 17], [42 \mapsto 0, 43 \mapsto 3, 17 \mapsto 4])$$

and suppose that x and y are client variables and that fl is a pointer to the free list of the module. It can be easily seen that the current heap consists of the module part – list of length one $h_p = [42 \mapsto 0, 43 \mapsto 3]$, and the remaining part $h_u = [17 \mapsto 4]$, which belongs to the client, while variables x and fl are aliases. Now, clearly $\neg(s,h)[[\![x{\rightarrow}nxt{=}x]\!]]\mathsf{wrong}$, but $(s,h_u)[[\![x{\rightarrow}nxt{=}x]\!]]\mathsf{wrong}$, and so by the above rule $(s,h), x{\rightarrow}nxt{=}x \rightsquigarrow^{i}_{(p,\eta)} \mathsf{av}$.

Notation for Command Meanings Parametrized by Modules. Finally, in describing results on refinement we will want to hold the $[\![\cdot]\!]$ semantics fixed, while varying the semantics of modules. If c is a command and (p,η) a module, then we write $c[(p,\eta)]$ for the relation of type

$$c[(p,\eta)] \subseteq \Sigma \times (\Sigma \cup \{\mathsf{wrong}\})$$

determined by the standard semantics $\rightsquigarrow_{(p,\eta)}$. With this notation we can write $c[(p,\eta)]$ and $c[(q,\mu)]$ for the same client command, but with different instantiations of the module semantics. Similarly, we write $c^i[(p,\eta)]$, with superscript i, for the relation

$$c^i[(p,\eta)] \subseteq \Sigma \times (\Sigma \cup \{\mathsf{wrong}, \mathsf{av}\})$$

determined by the instrumented semantics $\rightsquigarrow^{i}_{(p,\eta)}$.

7. Healthiness Conditions

Above we required that client and module operations be relations on states and states together with a faulty value

$$r \subseteq \Sigma \times (\Sigma \cup \{\mathsf{wrong}\}).$$

For our technical results on lifting simulations to go through these relations are required to satisfy three conditions. Fortunately, the conditions are conceptually as well as technically sensible. Two (the frame property and safety monotonicity [Yan01, YO02]) are familiar from work on separation logic, and one (contents independence) is new to this work.

Here are the three healthiness conditions that relations $r \subseteq (\Sigma \times (\Sigma \cup \{\mathsf{wrong}\} \uplus E))$ must satisfy, where E is a set of error values (the definition is parametrized to cover any such set, mainly because we will use two of them, $\{\mathsf{wrong}, \mathsf{av}\}$) and $\{\mathsf{wrong}\}$)).

- **Safety Monotonicity**: For all stacks s and heaps h and h_1 such that $h \# h_1$, if $\neg(s,h)[r]\mathsf{wrong}$, then $\neg(s, h \cdot h_1)[r]\mathsf{wrong}$.
- **Frame Property**: For all stacks s, s' and heaps h_0, h_1 and h' with $h_0 \# h_1$, if $\neg(s,h_0)[r]\mathsf{wrong}$ and $(s, h_0 \cdot h_1)[r](s', h')$ then there is a heap $h'_0 \sqsubseteq h'$ such that $h'_0 \# h_1$, $h'_0 \cdot h_1 = h'$ and $(s,h_0)[r](s', h'_0)$.
- **Contents Independence**: For all stacks s, s' and heaps h_0, h_1, h'_0 if

$$h_1 \# h_0 \ \wedge \ \neg(s,h_0)[r]\mathsf{wrong} \ \wedge \ (s, h_0 \cdot h_1)[r](s', h'_0 \cdot h_1)$$

  then we have that, for all h_2 satisfying $(\forall h.\ h \# h_1 \Longrightarrow h \# h_2)$,

$$(s, h_0 \cdot h_2)[r](s', h'_0 \cdot h_2).$$

The first two properties ensure that if heap h contains all the memory necessary for safe execution of the "command" r, then every computation from a bigger heap $h \cdot h_1$ is also safe. Moreover, such computation can be tracked from some computation from the smaller heap h. These properties ensure the soundness of the *frame rule* from separation logic

$$\frac{\{p\}c\{q\}}{\{p * r\}c\{q * r\}} \ \mathrm{Modifies}(c) \cap \mathrm{Free}(r) = \{\}.$$

Here, the "tight" interpretation of triples is assumed. Namely, $\{p\}c\{q\}$ is true just when for all (s,h), if $(s,h) \models p$ then

- $c, s, h \not\rightsquigarrow \mathsf{wrong}$, and
- $c, s, h \rightsquigarrow s', h'$ then $s', h' \models q$.

We refer to [Yan01, YO02] for further discussion of these properties.

The third property, contents independence, expresses that if command r is safe when executed from heap h_0, (i.e. it does not go wrong), the execution of r from a bigger heap $h_0 \cdot h_1$ does not look at the contents of heap h_1; it can only use the information that the heap memory in h_1 is allocated initially, and so heap h_1 can be replaced by any heap h_2 which contains at most all the locations allocated in h_1. In the RAM model, condition $(\forall h.\ h \# h_1 \Longrightarrow h \# h_2)$ boils down to $\mathsf{dom}(h_2) \subseteq \mathsf{dom}(h_1)$. The contents independence property is necessary in order to prove the lifting theorem.

As we have already mentioned above, the properties are necessary in order for our technical results to go through. In particular, the seemingly straightforward result stated in Lemma 4 in Section 9 requires a subtle use of all three properties. All of the subsequent results rest in part on this result, including the Lifting Theorem (Theorem 1).

Remark on Contents Independence versus Frame Property. At first glance, one might expect that contents independence follows from the frame property, but the following example shows that it does not. Let r be an action defined by

$$\begin{array}{lcl} (s,h)[r]\mathsf{wrong} & \stackrel{def}{\Longleftrightarrow} & 1 \notin \mathsf{dom}(h) \\ (s,h)[r](s,v) & \stackrel{def}{\Longleftrightarrow} & \left(\begin{array}{l} (2 \in \mathsf{dom}(h) \Longrightarrow \\ \quad (h(2) \neq 1 \Longrightarrow v = h) \ \vee \\ \quad (h(2) = 1 \Longrightarrow h = v \cdot [1 \mapsto -])) \ \vee \\ (2 \notin \mathsf{dom}(h) \Longrightarrow \\ \quad v = h \ \vee \ h = v \cdot [1 \mapsto -]) \end{array} \right). \end{array}$$

This "command" r satisfies both safety monotonicity and the frame property, but not contents independence. Note that $\mathsf{safe}(r, (s, [1 \mapsto 0]))$ and $2 \notin \mathsf{dom}([1 \mapsto 0])$. When we run "command" r in a state $(s, [1 \mapsto 0, 2 \mapsto 3])$ the output state is $(s, [1 \mapsto 0, 2 \mapsto 3])$. The contents independence property then ensures that if we replace $[2 \mapsto 3]$ by another heap with the domain which is the subset of $\mathsf{dom}([2 \mapsto 3]) = \{2\}$, for instance $[2 \mapsto 1]$, the output state should be $(s, [1 \mapsto 0, 2 \mapsto 1])$, which is here not the case. Here, instead, it is $(s, [2 \mapsto 1])$. The command r behaves differently depending on the contents of location 2.

Remark on Safety Monotonicity and Access Violation. If we were to conflate wrongness and access violation then safety monotonicity would fail. To see why, suppose that the resource invariant of the module is given by the predicate $(1 \mapsto 2) * (2 \mapsto 3)$, and suppose that the current heap is $h = [1 \mapsto 2]$. Then, the whole heap, h, must belong to the client and it is safe to perform a command $[1] := 42$, i.e. command $[1] := 42$ does not fault in state h. If we extend the current heap h with a disjoint heap $[2 \mapsto 3]$, the resource invariant becomes satisfied and command $[1] := 42$ now faults, which means that the heap-update command does not satisfy the safety monotonicity. Therefore, in accordance with the intuitive understanding, we keep av and wrong as two separate erroneous states.

We now collect the healthiness conditions together into a complete lattice, and verify that our semantics preserves them.

Definition 3 (General Local Action). A *general local action* is a relation that satisfies safety monotonicity, frame property and contents independence. We let GLAct denote the set of general local actions.

Lemma 1. When the set GLAct of general local actions is given the subset order, it becomes a complete lattice.

Proof. Note that it is sufficient to prove that $(\mathsf{GLAct}, \subseteq)$ has all the (possibly infinitary or nullary) joins. We will prove this sufficient condition by showing that GLAct is closed under arbitrary unions. For each family $\{r_i\}_{i \in I}$ of general local actions, this union of r_i's will become the join of the family.

Consider a family $\{r_i\}_{i \in I}$ of general local actions. Let $r = \bigcup_{i \in I} r_i$. Pick heaps h_0, h_1 such that

$$\neg(s, h_0)[r]\mathsf{wrong} \quad \wedge \quad h_0 \# h_1.$$

Then, by the definition of r, we have that

$$\forall i \in I.\ \neg(s, h_0)[r_i]\mathsf{wrong}. \tag{1}$$

Using this property, we will show that r satisfies the three healthiness conditions, so that it is a general local action. The first condition is the safety monotonicity, and it is $\neg(s, h_0 \cdot h_1)[r]\mathsf{wrong}$, equivalently,

$$\forall i \in I.\ \neg(s, h_0 \cdot h_1)[r_i]\mathsf{wrong}.$$

This condition follows from (1) and the safety monotonicity of r_i's. Next, we move on to the frame property of r. Consider (s', h') such that $(s, h_0 \cdot h_1)[r](s', h')$. By the definition of r, this means that

$$(s, h_0 \cdot h_1)[r_j](s', h') \text{ for some } j \in I. \tag{2}$$

But r_j is safe at (s, h_0) as shown in (1) and it satisfies the frame property. Thus, (2) implies the existence of some $h_0' \sqsubseteq h'$ satisfying the properties below:

$$h_0' \# h_1 \quad \wedge \quad h_0' \cdot h_1 = h' \quad \wedge \quad (s, h_0)[r_j](s', h_0').$$

Since the third property above entails $(s, h_0)[r](s', h_0')$, the above properties of h_0' show that h_0' is the heap required by the frame property of r. Finally, we show the contents independence of r. Let s', h_0' and h_2 be stack and heaps such that

$$(s, h_0 \cdot h_1)[r](s', h_0' \cdot h_1) \quad \wedge \quad (\forall h.\, h \# h_1 \Longrightarrow h \# h_2). \tag{3}$$

We need to prove that $(s, h_0 \cdot h_2)[r](s', h_0' \cdot h_2)$. By the definition of r, the first conjunt of (3) implies that $(s, h_0 \cdot h_1)[r_j](s', h_0' \cdot h_1)$ for some $j \in I$. But r_j is safe at (s, h_0) and it also satisfies the contents independence. Thus, by the choice of h_2 (i.e., the second conjunct of (3)), we also have that $(s, h_0 \cdot h_2)[r_j](s', h_0' \cdot h_2)$. By the definition of r again, this means that $(s, h_0 \cdot h_2)[r](s', h_0' \cdot h_2)$, which is the property of r that we are looking for. □

Lemma 2. All the primitive operations of the concrete user programming language given in Table 3 are general local actions. That is, for every $\mathsf{cl\_op}$ in the table, $[\![\mathsf{cl\_op}]\!]$ is a general local action.

Proof. All the commands satisfy safety monotonicity and frame property; the proof can be found in [YO02]. We only need to prove that all of them also satisfy contents independence. We consider only the heap update command as the similar reasoning may be applied to other basic commands.

Let r be the semantics $[\![[E_1]{:=}E_2]\!]$ of the update command $[E_1] := E_2$. Consider stacks s, s' and heaps

h_0, h_0' and h_1, such that $\neg(s, h_0)[r]\mathsf{wrong}$ and $(s, h_0 \cdot h_1)[r](s', h_0' \cdot h_1)$. The frame property and the cancellativity (i.e., injectivity) of $\cdot$ yield $(s, h_0)[r](s, h_0')$. The definition of a command r then implies that $[\![E_1]\!] \in \mathsf{dom}(h_0)$ and $h_0' = h_0[[\![E_1]\!] \mapsto [\![E_2]\!]]$. Let h_2 be any heap such that $\mathsf{dom}(h_2) \subseteq \mathsf{dom}(h_1)$. Then,

$$h_0' \cdot h_2 = h_0[[\![E_1]\!] \mapsto [\![E_2]\!]] \cdot h_2 = (h_0 \cdot h_2)[[\![E_1]\!] \mapsto [\![E_2]\!]].$$

By the definition of a command $[E_1] := E_2$, it follows that $(s, h_0 \cdot h_2)[[E_1] := E_2](s', h_0' \cdot h_2)$, which is what we wanted to prove. □

Lemma 3. For every command c that can be generated by the grammar of the general programming language, and every module (p, η), the relations $c[(\eta, p)]$ and $c^i[(\eta, p)]$ (determined by the normal operational semantics and the instrumented semantics of c, respectively) are general local actions, provided that the primitive client and module operations are.

Proof. Consider a module (p, η). Here we only prove that $c^i[(\eta, p)]$ is a general local action, because similar reasoning can be applied for $c[(\eta, p)]$. Our proof is conducted by induction on the structure of the command c. By the assumption of the lemma, all the primitive client and module operations are general local actions. To prove that all the compound commands of the language are general local actions, we only prove that they preserve contents independence, as it is already known that they preserve the frame property and safety monotonicity [YO02]. So in each case we assume the antecedent of contents independence and prove the consequent.

In order to simplify notation, we write $[c^i]$ instead of $c^i[(p, \eta)]$, and $\rightsquigarrow^i$ instead of $\rightsquigarrow^i_{(p,\eta)}$. Let s, s' and h_0, h_0' and h_1 be such that $\neg(s, h_0)[c^i]\mathsf{wrong}$ and $(s, h_0 \cdot h_1)[c^i](s', h_0' \cdot h_1)$.

Let c be a sequential composition of commands c_1 and c_2. From the antecedent of contents independence we have $\neg(s, h_0)[(c_1; c_2)^i]$ wrong and therefore

$$\neg(s, h_0)[c_1^i]\mathsf{wrong} \ \wedge \tag{4}$$

$$(\forall s'', h_0''.\ (s, h_0)[c_1^i](s'', h_0'') \implies \neg(s'', h_0'')[c_2^i]\mathsf{wrong}). \tag{5}$$

The definition of the operational semantics of sequential composition and the assumption $(s, h_0 \cdot h_1)[c^i](s', h_0' \cdot h_1)$ ensure that there exists a state (s'', h''), such that

$$(s, h_0 \cdot h_1)[c_1^i](s'', h'') \ \wedge \tag{6}$$

$$(s'', h'')[c_2^i](s', h_0' \cdot h_1). \tag{7}$$

Using the fact that c_1 satisfies the frame property and 4 and 6, we conclude that there exists a heap h_0'' such that

$$(s, h_0)[c_1^i](s'', h_0'') \ \wedge \ h'' = h_0'' \cdot h_1, \text{ i.e. } (s, h_0 \cdot h_1)[c_1^i](s'', h_0'' \cdot h_1). \tag{8}$$

From 5 and 8 we get that $\neg(s'', h_0'')[c_2^i]\mathsf{wrong}$. So far we know that the following hold:

$$\neg(s, h_0)[c_1^i]\mathsf{wrong} \ \wedge \ (s, h_0 \cdot h_1)[c_1^i](s'', h_0'' \cdot h_1) \tag{9}$$

$$\text{and } \neg(s'', h_0'')[c_2^i]\mathsf{wrong} \ \wedge \ (s'', h_0'' \cdot h_1)[c_2^i](s', h_0' \cdot h_1). \tag{10}$$

We now exploit the fact that each of the commands c_1 and c_2 satisfies contents independence. Consider a heap h_2 such that $\forall h.h\#h_1 \implies h\#h_2$. From 9 and 10, we have that

$$(s, h_0 \cdot h_2)[c_1^i](s'', h_0'' \cdot h_2) \quad \text{and} \quad (s'', h_0'' \cdot h_2)[c_2^i](s', h_0' \cdot h_2)$$

hold, respectively. By the definition of the semantics of sequential composition this is equivalent to

$$(s, h_0 \cdot h_2)[(c_1; c_2)^i](s', h_0' \cdot h_2).$$

Let c be **if** B **then** $\mathsf{c_1}$ **else** $\mathsf{c_2}$. Consider the case when $[\![B]\!]_s = \mathsf{true}$. By the definition of the **if**−**then**−**else** statement,

$$\neg(s, h_0)[c_1^i]\mathsf{wrong} \qquad \text{and} \qquad (s, h_0 \cdot h_1)[c_1^i](s', h_0' \cdot h_1).$$

Similarly, when $[\![B]\!]_s = \mathsf{false}$, we have that

$$\neg(s, h_0)[c_2^i]\mathsf{wrong} \qquad \text{and} \qquad (s, h_0 \cdot h_1)[c_2^i](s', h_0' \cdot h_1).$$

Let h_2 be an arbitrary state such that $\forall h.\, h\#h_1 \implies h\#h_2$. Then, since c_1 and c_2 both obey the contents independence property, it follows that

$$[\![B]\!]_s = \mathsf{true} \implies (s, h_0 \cdot h_2)[c_1^i](s', h_0' \cdot h_2)$$
$$[\![B]\!]_s = \mathsf{false} \implies (s, h_0 \cdot h_2)[c_2^i](s', h_0' \cdot h_2).$$

Since h_2 was an arbitrary heap with $(\forall h.\, h\#h_1 \implies h\#h_2)$, this is true for all such heaps, and hence, the **if** $-$ **then** $-$ **else** command satisfies the contents independence property.

For the while-case, suppose that

$$\neg(s, h_0)[(\textbf{while } B \textbf{ do } c)^i]\mathsf{wrong} \quad \text{and} \quad (s, h_0 \cdot h_1)[(\textbf{while } B \textbf{ do } c)^i](s', h_0' \cdot h_1).$$

Then, by the definition of $[c^i]$ (i.e., the relation $c^i[(p, \eta)]$), this is equivalent to

$$\textbf{while } B \textbf{ do } c, (s, h_0) \not\rightsquigarrow^i \mathsf{wrong} \quad \text{and} \quad \textbf{while } B \textbf{ do } c, (s, h_0 \cdot h_1) \rightsquigarrow^i (s', h_0' \cdot h_1).$$

We will prove that for any heap h_2 such that $\forall h.\, h\#h_1 \implies h\#h_2$, we also have

$$\textbf{while } B \textbf{ do } c, (s, h_0 \cdot h_2) \quad \rightsquigarrow^i \quad (s', h_0' \cdot h_2). \tag{11}$$

From here, it will follow, again by the definition of $[c^i]$, that

$$(s, h_0 \cdot h_2)[(\textbf{while } B \textbf{ do } c)^i](s', h_0' \cdot h_2).$$

To get (11), we use the induction on the number of while-unwinding in the derivation of **while** B **do** $c, (s, h_0 \cdot h_1) \rightsquigarrow^i (s', h_0' \cdot h_1)$. If the length of the derivation is 0, then $[\![B]\!]_s = \mathsf{false}$ and so we have $s = s'$ and $h_0 \cdot h_1 = h_0' \cdot h_1$, the second of which gives $h_0 = h_0'$ by the cancellativity (i.e., injectivity) of $\cdot$. Then, since the stack is unchanged in the new state, i.e. $[\![B]\!]_s = \mathsf{false}$ again, by the rules of semantics for **while** command, we have that

$$\textbf{while } B \textbf{ do } c, \ (s, h_0 \cdot h_2) \quad \rightsquigarrow^i \quad (s, h_0 \cdot h_2).$$

Suppose now that the length of the derivation is $n + 1$ and that the claim is true for all the derivations of length n or less. Then, by the rule of semantics used to obtain this derivation, we have the following.

$$[\![B]\!]_s = \mathsf{true} \quad \text{and} \quad c; (\textbf{while } B \textbf{ do } c), \ (s, h_0 \cdot h_1) \rightsquigarrow^i (s', h_0' \cdot h_1).$$

Then, there exists a state (s'', h''), such that $c, (s, h_0 \cdot h_1) \rightsquigarrow^i (s'', h'')$, i.e. $(s, h_0 \cdot h_1)[c^i](s'', h'')$, and **while** B **do** $c, (s'', h'') \rightsquigarrow^i (s', h_0' \cdot h_1)$. We apply similar reasoning as in case of the sequential composition and obtain some heap h_0'', such that $h'' = h_0'' \cdot h_1$. Now, we can apply the outer induction hypothesis to the command c and the inner induction hypothesis to the derivation **while** B **do** $c, (s'', h_0'' \cdot h_1) \rightsquigarrow^i (s', h_0' \cdot h_1)$, as it is of length n. Therefore, we have that for any heap h_2 such that $\forall h.\, h\#h_1 \implies h\#h_2$,

$$c, (s, h_0 \cdot h_2) \rightsquigarrow^i (s'', h_0'' \cdot h_2) \quad \text{and} \quad \textbf{while } B \textbf{ do } c, (s'', h_0'' \cdot h_2) \rightsquigarrow^i (s', h_0' \cdot h_2),$$

which will give us the desired result. □

In our theory we will also make use of the following extensions of the language operations. Let op denote an operation on the set of the states, i.e. $op \subseteq \Sigma \times (\Sigma \cup \{\mathsf{wrong}, \mathsf{av}\})$. Let E denote a set of exceptions. For example, $E = \{\mathsf{wrong}\}$ or $E = \{\mathsf{wrong}, \mathsf{av}\}$. Then, by op_E, we denote the operation $op_E \subseteq (\Sigma \uplus E) \times (\Sigma \uplus E)$, such that $op_E = op \uplus \{(e, e) \mid e \in E\}$. For example, for $E = \{\mathsf{wrong}, \mathsf{av}\}$, $op_E = op \uplus \{(\mathsf{wrong}, \mathsf{wrong}), \ (\mathsf{av}, \mathsf{av})\}$. When op is $c[(p, \eta)]$ or $c^i[(p, \eta)]$, we will write $c_E[(p, \eta)]$ and $c_E^i[(p, \eta)]$, instead of $(c[(p, \eta)])_E$ and $(c^i[(p, \eta)]_E$. This is in order to avoid the unnecessary complication in notations.

In summary, we require the interpretations of all client and module operations, $[\![\mathsf{cl\_op}]\!]$ and $\eta(\mathsf{mod\_op})$, to satisfy the three healthiness conditions stated at the beginning of this section.

8. Growing Coupling Relations

Having now given a semantics to our language parametrized by modules, we can move on to the use of binary relations to connect different modules. We cannot use arbitrary relations, but rather use ones satisfying two restrictions.

The first requirement is that such relations, which we call *coupling* relations, connect the internal states of two different modules as represented by their resource invariants.

Definition 4. A *coupling* relation $R \subseteq \Sigma \times \Sigma$ between modules (p, η) and (q, ϵ) is a binary relation such that

- the domain and range of R are resource invariants p and q:

$$(s_1, h_1)[R](s_2, h_2) \implies (s_1, h_1) \in p \ \wedge \ (s_2, h_2) \in q,$$

- R does not depend on the client's stack:

$$(s_{1c} \cup s_{1m}, h_1)[R](s_{2c} \cup s_{2m}, h_2) \iff (s'_{1c} \cup s_{1m}, h_1)[R](s'_{2c} \cup s_{2m}, h_2).$$

We need a second requirement on relations for the simulation method to be sound: we must restrict the relations to be "growing", where concrete states use more (or equal) resource than abstract. The condition is not completely unintuitive, as often a refinement will produce a more detailed data structure, which uses more heap than in the abstract. However, there is also a technical need for the growing condition. One of the main results needed for the lifting theorem is that a client operation simulates itself (Lemma 4 in the next section). Perhaps surprisingly, this result fails without growing relations, as we show with a counterexample at the end of the next section.

Definition 5. A relation $R \subseteq \Sigma \times \Sigma$ is *growing* if and only if for all states (s, h) and (s', h')

$$(s, h)[R](s', h') \implies (\forall h''.\ h''\#h' \implies h''\#h).$$

Intuitively, if R relates the internal states of two modules, then the above condition means that one module uses more memory than the other.

We have already seen the clause for growing relations in the definition of contents independence, and we remark again that in the RAM model the condition is equivalent to

$$(s, h)[R](s', h') \implies \mathsf{dom}(h) \subseteq \mathsf{dom}(h') \ .$$

Example. We illustrate coupling relations with our running example. For the two interpretations of the toy memory manager module from Table 5, we give the corresponding coupling relation. It is

$$R = \{((s_1, h_1), (s_2, h_2)) \mid \exists \alpha.\, (s_1, h_1) \models \mathsf{Set}(\alpha, S) \ \text{ and } \ (s_2, h_2) \models \mathsf{list}(\alpha, fl)\}.$$

Notice that the $\exists\alpha$ is in a position where the same α is used in $\mathsf{Set}(\alpha, S)$ and in $\mathsf{list}(\alpha, fl)$. This formalizes the "are precisely the elements of" aspect of the informal description from Section 2, which was

- `Simulation Relation`: fl points to an acyclic linked list whose nodes are the elements of set S.

Constructions of Relations. We define several ways of constructing relations. First, the separating conjunction for predicates (unary relations) can be used to partition the state between a client and a single representation of a module. We can do the same thing for simulation relations in place of unary predicates by defining the separating conjunction of relations. For two binary relations $R, S \subseteq \Sigma \times \Sigma$ on states, we define their *separating conjunction* [RY03] as

$$R * S = \left\{ ((s_1, h_1), (s_2, h_2)) \;\middle|\; \begin{array}{c} \exists h'_1, h''_1, h'_2, h''_2.\ \ h_1 = h'_1 \cdot h''_1 \ \wedge h_2 = h'_2 \cdot h''_2 \ \wedge \\ (s_1, h'_1)[R](s_2, h'_2) \ \wedge \ (s_1, h''_1)[S](s_2, h''_2) \end{array} \right\}$$

The separating conjunction of binary relations gives us a way to *expand* a coupling relation between two modules, giving us a relation that applies to client state as well. For instance, for two modules (p, η) and (q, ϵ), we expand relation R between them to a relation $R * \mathsf{Id}$, where $\mathsf{Id} \subseteq \Sigma \times \Sigma$ denotes the identity relation on the client's parts of states:

$$\begin{array}{l} \forall s_{1c}, s_{2c} \in S_c.\ \forall s_{1m}, s_{2m} \in S_m.\ \forall h_1, h_2 \in H. \\ \qquad (s_{1c} \cup s_{1m}, h_1)[\mathsf{Id}](s_{2c} \cup s_{2m}, h_2) \iff (s_{1c} = s_{2c} \ \wedge \ h_1 = h_2). \end{array}$$

The R part of $R * \mathsf{Id}$ requires the modules' internals to be related by R, where the Id part requires that the client parts of the states be equal. This can be used to express that the client program does not change when we replace one module by another.

Note that when the identity relation Id is composed by separating conjunction with a growing relation R, the resulting relation $R * \mathsf{Id}$ is also growing.

Trumping Simulations. Finally, a very special kind of simulation relations, trumping simulations, plays an essential role in this work. Technically, these relations say that abstract fault is related to anything. The idea is that the specifier is responsible for showing that an abstract program does not fault: if the abstract program faults, then all bets are off.

If E is a set (thought of as a set of exceptions, or faulty states) then the relation

$$R_E \subseteq (\Sigma \uplus E) \times (\Sigma \uplus E)$$

behaves as R on the Σ part, and allows an element of the leftmost E to be related to anything. Formally, let R denote a relation on pairs of states, that is $R \subseteq \Sigma \times \Sigma$ and let E denote a set of exceptions (faulty states) disjoint from Σ. Then,

$$R_E \;=\; R \cup \{(e, d) \mid e \in E \;\wedge\; d \in \Sigma \uplus E\}.$$

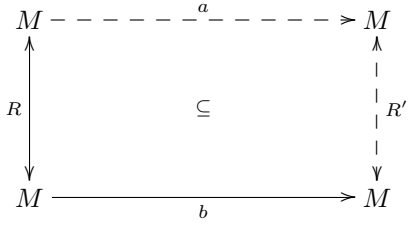
For example, in $R_{\{\mathsf{wrong}\}}$ we have that $(\mathsf{wrong}, d) \in R_E$ no matter what d is.

Though technically straightforward in definition, the decision to employ trumping simulations has played a central simplifying role in this work. By using somewhat unusual simulation relations, we are able to use standard conditions on what counts as a simulation (Definition 6 in the next section) to reason about low-level pointer programs.

9. Simulation and the Instrumented Lifting Theorem

Forward simulation method is a traditional way of proving data refinement. Its main component is a simulation relation which provides a correlation between the states of the "abstract" program and its concrete counterpart. The method then ensures that whatever step of the computation the concrete program makes starting from some state, then starting from a related state the abstract program can perform the same step, again, up to the simulation relation.

Definition 6. Let M denote a set of states and let R and R' denote relations between the pairs of states, i.e. $R, R' \subseteq M \times M$. We say that a diagram

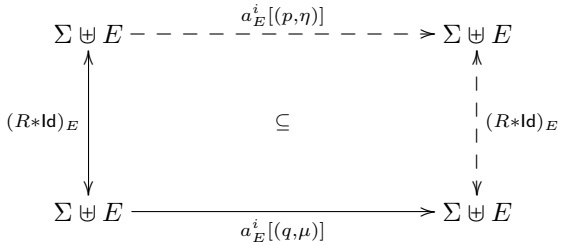


holds for operations $a, b \subseteq M \times M$ if and only if for all states m_1, m_2 and m_2', such that $m_1[R]m_2$ and $(m_2, m_2') \in b$, then there exists a state m_1', such that $(m_1, m_1') \in a$ and $m_1'[R']m_2'$. We write this condition more succinctly as $R; b \subseteq a; R'$. When this diagram holds we say that operation b simulates operation a.

We are now in a position to state and prove our main results concerning simulation. First, any primitive client operation simulates itself. (This simple-sounding result was in fact a thorny one that drove our theory. It required both growing relations and the formulation of the contents-independence property, as will be seen from the proof.)

Lemma 4. Let (p, η) and (q, μ) be two modules such that $\mathsf{dom}(\mu) = \mathsf{dom}(\eta)$ and let $R * \mathsf{Id}$ be a growing coupling relation between them. Also, let E denote the set of exceptions containing wrong and av, i.e.,

$E = \{\mathsf{wrong}, \mathsf{av}\}$. Then, for all primitive client operations a, we have that the following diagram holds.



Proof. Let $d_1, d_2, d_2' \in \Sigma \uplus E$ be such that

$$d_1[(R * \mathsf{Id})_E]d_2 \quad \text{and} \quad d_2[a^i_E[(q,\mu)]]d_2'.$$

We are required to find d_1' such that

$$d_1'[(R * \mathsf{Id})_E]d_2' \quad \text{and} \quad d_1[a^i_E[(p,\eta)]]d_1'.$$

Note that by the definition of $(R * \mathsf{Id})_E$, there are several possibilities for the values of d_1 and d_2.

Firstly, suppose that d_1 is either wrong or av. The definition of $a^i_E[(p,\eta)]$ ensures that d_1 itself is an output of $a^i_E[(p,\eta)]$ for the input d_1. Also, the definition of $(R * \mathsf{Id})_E$ guarantees that d_1 and d_2' are related by it. Thus, in this case, d_1 is the desired d_1'.

Secondly, suppose that d_1 is not faulty, i.e., it is neither wrong nor av. Then, by the definition of $(R{*}\mathsf{Id})_E$, state d_2 should not be faulty, either. Thus, we can denote states d_1 and d_2 as $d_1 = (s_1, h_1)$ and $d_2 = (s_2, h_2)$. Since (s_1, h_1) and (s_2, h_2) are related by $R * \mathsf{Id}$, they can be decomposed according to the separating conjunction as $(s_1, h_1) = (s_c \cup s_{1m}, h_c \cdot h_p)$ [3] and $(s_2, h_2) = (s_c \cup s_{2m}, h_c \cdot h_q)$, for some stacks $s_c \in S_c$ and $s_{1m}, s_{2m} \in S_m$, and some heaps h_p, h_q, h_c such that $(s_c \cup s_{1m}, h_p)[R](s_c \cup s_{2m}, h_q)$. Further, there are two possibilities.

Suppose that

$$a, (s_2, h_2) \rightsquigarrow^i_{(q,\mu)} \mathsf{wrong} \quad \text{or} \quad a, (s_2, h_2) \rightsquigarrow^i_{(q,\mu)} \mathsf{av}.$$

In this case, we can prove the lemma by showing that

$$a, (s_1, h_c) \rightsquigarrow^i_{(p,\eta)} \mathsf{wrong}. \tag{12}$$

The reason for this is as follows. The above computation of a, if exists, implies

$$a, (s_1, h_1) \rightsquigarrow^i_{(p,\eta)} \mathsf{wrong} \quad \text{or} \quad a, (s_1, h_1) \rightsquigarrow^i_{(p,\eta)} \mathsf{av},$$

which means that

$$(s_1, h_1)\big[a^i[(p,\eta)]\big]\mathsf{wrong} \quad \text{or} \quad (s_1, h_1)\big[a^i[(p,\eta)]\big]\mathsf{av}.$$

This shows that the desired d_1' exists and that it is wrong or av, because by the definition of $(R * \mathsf{Id})_E$, both wrong and av are related to every element in $\Sigma \cup \{\mathsf{wrong}, \mathsf{av}\}$. Now, let's go back to the proof of 12 above. By the rules of semantics instrumented by (q,μ), the derivation $a, (s_2, h_2) \rightsquigarrow^i_{(q,\mu)} \mathsf{wrong}$ is possible only when $(s_2, h_2)[[\![a]\!]]\mathsf{wrong}$. But then, the safety monotonicity gives

$$(s_2, h_c)[[\![a]\!]]\mathsf{wrong}.$$

The same relationship also holds, when $a, (s_2, h_2) \rightsquigarrow^i_{(q,\mu)} \mathsf{av}$, because one condition of the rule for generating av (in the semantics instrumented by (q,μ)) is precisely that relationship. From this $(s_2, h_c)[[\![a]\!]]\mathsf{wrong}$, it follows that $(s_1, h_c)[[\![a]\!]]\mathsf{wrong}$, because s_2 and s_1 differ only for module variables and a is a primitive client operation. Now, the rules for the semantics say that, since $(s_1, h_c)[[\![a]\!]]\mathsf{wrong}$,

$$a, (s_1, h_c) \rightsquigarrow^i_{(p,\eta)} \mathsf{wrong}$$

as required.

[3] Recall that the stacks consist of two parts: the client part s_c and the module part s_m

Finally, suppose that

$$a, (s_2, h_2) \not\rightsquigarrow^i_{(q,\mu)} \mathsf{wrong}, \quad a, (s_2, h_2) \not\rightsquigarrow^i_{(q,\mu)} \mathsf{av} \quad \text{and} \quad a, (s_2, h_2) \rightsquigarrow^i_{(q,\mu)} (s_2', h_2').$$

Then, by the rules for the semantics,

$$a, (s_2, h_c) \not\rightsquigarrow^i_{(q,\mu)} \mathsf{wrong}.$$

We can apply the frame property for a, and so there exists a heap h_c' such that $h_2' = h_c' \cdot h_q$ and $a, (s_2, h_c) \rightsquigarrow^i_{(q,\mu)} (s_c' \cup s_{2m}', h_c')$, where s_c' and s_{2m}' are the client and module parts of s_2'. Furthermore, since the primitive client command a does not modify module variables, $s_{2m}' = s_{2m}$.

Let $(s_1', h_1') = (s_c' \cup s_{1m}, h_c' \cdot h_p)$. We claim this is the state we are looking for in order to have the simulation condition to be fulfilled. Firstly, state (s_1', h_1') is defined. R is growing, so for all heaps h, if $h \# h_q$ then $h \# h_p$, and since we already know that $h_c' \# h_q$, it follows that $h_c' \cdot h_p$ is defined. Secondly, states (s_1', h_1') and (s_2', h_2') are related by $R * \mathsf{Id}$. Finally, we need to show that $(s_1, h_1)[a^i_E[(p, \eta)]](s_1', h_1')$, equivalently,

$$a, (s_1, h_1) \rightsquigarrow^i_{(p,\eta)} (s_1', h_1').$$

The following things hold, by what we have proved so far.

1. $\neg(s_c \cup s_{2m}, h_c)[[\![a]\!]]\mathsf{wrong}$
2. $(s_c \cup s_{2m}, h_c \cdot h_q)[[\![a]\!]](s_c' \cup s_{2m}, h_c' \cdot h_q)$
3. $\forall h.\ h \# h_q \Rightarrow h \# h_p$

By the contents independence property of a, from 1, 2 and 3, we have that $(s_c \cup s_{2m}, h_c \cdot h_p)[[\![a]\!]](s_c' \cup s_{2m}, h_c' \cdot h_p)$. Now, by the fact that primitive client operations may access only client variables, it follows that $(s_c \cup s_{1m}, h_c \cdot h_p)[[\![a]\!]](s_c' \cup s_{1m}, h_c' \cdot h_p)$. Hence, $a, (s_1, h_1) \rightsquigarrow^i_{(p,\eta)} (s_1', h_1')$. □

Counterexample: The Need for Growing Relations. As we already mentioned, we make use of a special kind of relations in conjunction with the traditional forward simulation method to ensure its soundness. We consider only relations extended so to encounter for the faulty states wrong and av, in accordance with our instrumented semantics. However, even with this extension the problem of soundness remains. The cause of unsoundness is that for some client operations we have the situation where an operation fails to simulate itself. Namely, there are cases where if we run the client operation in two different module environments, even when the modules simulate one another, the client operation does not preserve the refinement relation. This is especially true for allocation. Refinement relation preservation by all the client operations is crucial for the soundness of the method. To prevent this complication we need to impose certain restrictions on the simulation relation. Namely, we have to make sure that for the related pairs of states, the abstract state is "smaller" then the concrete one, that is, we need to use only growing relations. To see why a bigger abstract state in a pair of related states might cause problems, consider the following example.

For simplicity, suppose that the heap locations are integers, that the states consist only of heaps and that the alloc() operation simply returns some location that is not in the current heap. Let simulation relation R contain a pair $([2 \mapsto 3], [])$. The abstract state contains exactly one location with address 2 and contents 3, and the concrete state is empty. Suppose now that we have two states related by $R * \mathsf{Id}$, $h_1 = [1 \mapsto 2, 2 \mapsto 3]$ and $h_2 = [1 \mapsto 2]$, and we run the alloc() operation on the concrete state.

$$h_1 [R * \mathsf{Id}] h_2 \quad \text{and} \quad \mathsf{alloc}(), h_2 \rightsquigarrow^i_{(p,\eta)} h_2 \cdot [2 \mapsto 0]$$

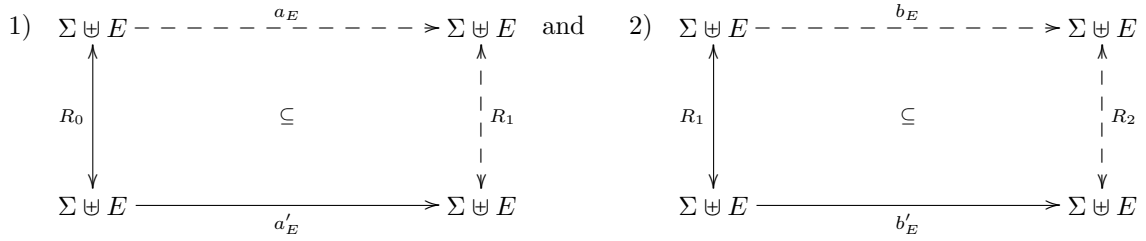
The operation alloc() executed on state h_2 might return location 2, as it is available in the state h_2. However, there is no such location that can be returned by alloc() when executed in state h_1 and relation $R * \mathsf{Id}$ be reestablished at the same time, i.e.,

$$\mathsf{alloc}(), h_1 \rightsquigarrow^i_{(p,\eta)} ???, \quad \text{such that} \quad ??? \, [R * \mathsf{Id}] \, (h_2 \cdot [2 \mapsto 0]).$$

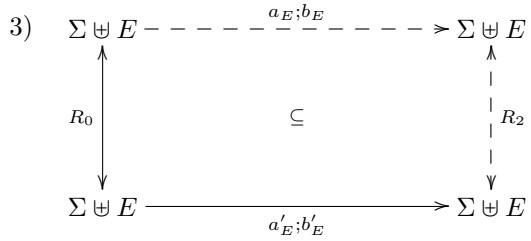
End of Counterexample.

Next, forward simulation is compositional with respect to the sequential composition. For $a, b \subseteq X \times Y$ and $b \subseteq Y \times Z$, let $a; b$ be their relational composition $\{(x, z) \mid \exists y.\, (x, y) \in a \ \wedge \ (y, z) \in b\}$.

Lemma 5. Let E denote the set of exceptions $\{\mathsf{wrong}, \mathsf{av}\}$. For all relations $R_0, R_1, R_2 \subseteq (\Sigma \uplus E) \times (\Sigma \uplus E)$ and operations $a_E, a'_E, b_E, b'_E \subseteq (\Sigma \uplus E) \times (\Sigma \uplus E)$, if diagrams



hold, then the diagram



also holds.

Proof. Let diagrams 1) and 2) hold. Let states $d_1, d_2, d''_2 \in \Sigma \uplus E$ be such that $d_1[R_0]d_2$ and $d_2[a'_E; b'_E]d''_2$. By the definition of relational composition, this means that there exists a state d'_2 such that $d_2[a'_E]d'_2$ and $d'_2[b'_E]d''_2$. From diagram 1) it follows that there exists a state d'_1 such that

$$d_1[a_E]d'_1 \quad \wedge \quad d'_1[R_1]d'_2 \quad \wedge \quad d'_2[b'_E]d''_2.$$

Using the assumption that diagram 2) holds this can be further transformed into

$$\exists d'_1 d''_1. \quad d_1[a_E]d'_1 \quad \wedge \quad d'_1[b_E]d''_1 \quad \wedge \quad d''_1[R_2]d''_2$$

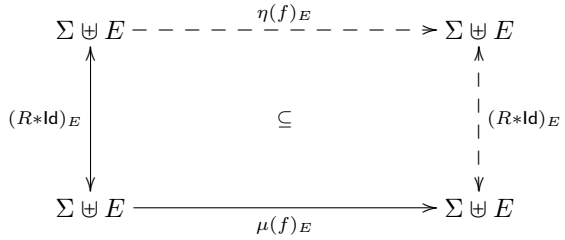
i.e.

$$\exists d''_1. \quad d_1[a_E; b_E]d''_1 \quad \wedge \quad d''_1[R_2]d''_2.$$

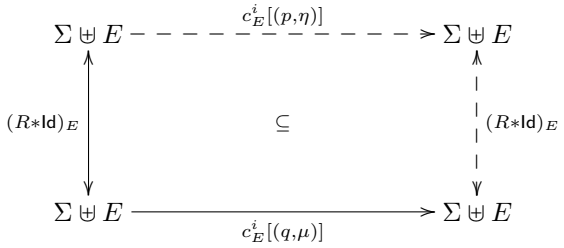
which completes the proof that the diagram 3) holds. □

We can finally state our main result.

Theorem 1 (Instrumented Lifting theorem). Let (p, η) and (q, μ) be two modules such that $\mathsf{dom}(\mu) = \mathsf{dom}(\eta)$ and let $R * \mathsf{Id}$ be a growing relation between them. Also, let E denote the set of exceptions consisting of faulting states wrong and av, i.e, $E = \{\mathsf{wrong}, \mathsf{av}\}$. If for all $f \in \mathsf{dom}(\mu)$ the diagram



holds, then for all contexts $c[\cdot]$ the following diagram holds, too.



Proof. We prove the theorem by induction on the structure of command c. When c is a module operation, the theorem follows from the assumption. When c is a primitive client operation, the theorem holds because of Lemma 4. When c is a sequential composition, the induction step goes through because of Lemma 5.

Let $c \equiv \textbf{if } B \textbf{ then } c_1 \textbf{ else } c_2$ and let states d_1 and d_2 be such that $d_1[(R * \mathsf{Id})_E]d_2$. There are two possibilities. The first possibility is that $d_1[c_E^i[(p,\eta)]]d'$ for some $d' \in E$. If this is the case, we can prove this theorem using d'. Note that element d' is related to every element in $(\Sigma \uplus E)$ by $(R * \mathsf{Id})_E$, because it is in E. Thus, for all d_2' satisfying $d_2[c_E^i[(q,\mu)]]d_2'$, if we choose d' as a required element by this theorem, we have that $d_1[c_E^i[(p,\eta)]]d'$ and $d'[(R * \mathsf{Id})_E]d_2'$, as desired.

The other case is that there are no $d' \in E$ satisfying $d_1[c_E^i[(p,\eta)]]d'$. In this case, d_1 cannot be an element in E, because $c_E^i[(p,\eta)]$ relates every element in E to itself. Furthermore, since $d_1[(R * \mathsf{Id})_E]d_2$, d_2 cannot be an element in E, either. So, we can write $d_1 = (s_1, h_1)$ and $d_2 = (s_2, h_2)$ for some states (s_1, h_1) and (s_2, h_2).

We now show that the concrete command cannot produce an element in E. Suppose that it can. Then, we derive the contradiction as follows:

$(s_2, h_2)\big[c_E^i[(q,\mu)]\big]\mathsf{wrong} \quad \vee \quad (s_2, h_2)\big[c_E^i[(q,\mu)]\big]\mathsf{av}$

$\implies$ (by the definition of $c_E^i[(q,\mu)]$)

$c, (s_2, h_2) \rightsquigarrow^i_{(q,\mu)} \mathsf{wrong} \quad \vee \quad c, (s_2, h_2) \rightsquigarrow^i_{(q,\mu)} \mathsf{av}$

$\implies$ (by the semantics of the if-then-else statement)

$\big([\![B]\!]_{s_2} = \mathsf{true} \ \wedge \ (c_1, (s_2, h_2) \rightsquigarrow^i_{(q,\mu)} \mathsf{wrong} \ \vee \ c_1, (s_2, h_2) \rightsquigarrow^i_{(q,\mu)} \mathsf{av})\big) \ \vee$
$\big([\![B]\!]_{s_2} = \mathsf{false} \ \wedge \ (c_2, (s_2, h_2) \rightsquigarrow^i_{(q,\mu)} \mathsf{wrong} \ \vee \ c_2, (s_2, h_2) \rightsquigarrow^i_{(q,\mu)} \mathsf{av})\big)$

$\implies$ (by ind. hypo., $(s_1, h_1)[(R * \mathsf{Id})_E](s_2, h_2)$ and the fact that B has no module vars)

$\big([\![B]\!]_{s_1} = \mathsf{true} \ \wedge \ (c_1, (s_1, h_1) \rightsquigarrow^i_{(p,\eta)} \mathsf{wrong} \ \vee \ c_1, (s_1, h_1) \rightsquigarrow^i_{(p,\eta)} \mathsf{av})\big) \ \vee$
$\big([\![B]\!]_{s_1} = \mathsf{false} \ \wedge \ (c_2, (s_1, h_1) \rightsquigarrow^i_{(p,\eta)} \mathsf{wrong} \ \vee \ c_2, (s_1, h_1) \rightsquigarrow^i_{(p,\eta)} \mathsf{av})\big)$

$\implies$ (by the semantics of the if-then-else statement)

$c, (s_1, h_1) \rightsquigarrow^i_{(p,\eta)} \mathsf{wrong} \quad \vee \quad c, (s_1, h_1) \rightsquigarrow^i_{(p,\eta)} \mathsf{av}.$

$\implies$ (since there are no $d' \in E$ with $d_1[c_E^i[(p,\eta)]]d'$)

$\mathsf{false}.$

We will complete the proof of this case by showing the theorem for all normal output states (s_2', h_2') of the concrete command. Let state (s_2', h_2') be such that $(s_2, h_2)\big[c_E^i[(q,\mu)]\big](s_2', h_2')$. Then,

$(s_2, h_2)\big[c_E^i[(q,\mu)]\big](s_2', h_2')$

$\iff$ (by the definition of $c_E^i[(q,\mu)]$)

$c, (s_2, h_2) \rightsquigarrow^i_{(q,\mu)} (s_2', h_2')$

$\implies$ (by the semantics of the if-then-else statement)

$([\![B]\!]_{s_2} = \mathsf{true} \ \wedge \ c_1, (s_2, h_2) \rightsquigarrow^i_{(q,\mu)} (s_2', h_2')) \ \vee \ ([\![B]\!]_{s_2} = \mathsf{false} \ \wedge \ c_2, (s_2, h_2) \rightsquigarrow^i_{(q,\mu)} (s_2', h_2'))$

$\Longrightarrow$ (by ind. hypo., $(s_1,h_1)[(R*\mathsf{Id})_E](s_2,h_2)$ and the fact that B has no module vars)

$\big(\exists(s_1',h_1').\ [\![B]\!]_{s_1} = \mathsf{true}\ \wedge\ c_1,(s_1,h_1) \rightsquigarrow^i_{(p,\eta)} (s_1',h_1')\ \wedge\ (s_1',h_1')[(R*\mathsf{Id})_E](s_2',h_2')\big)\ \vee$
$\big(\exists(s_1',h_1').\ [\![B]\!]_{s_1} = \mathsf{false}\ \wedge\ c_2,(s_1,h_1) \rightsquigarrow^i_{(p,\eta)} (s_1',h_1')\ \wedge\ (s_1',h_1')[(R*\mathsf{Id})_E](s_2',h_2')\big)$

$\Longleftrightarrow$ (by the rules of classical logic)

$\exists(s_1',h_1').\ \big([\![B]\!]_{s_1} = \mathsf{true}\ \wedge\ c_1,(s_1,h_1) \rightsquigarrow^i_{(p,\eta)} (s_1',h_1')\ \wedge\ (s_1',h_1')[(R*\mathsf{Id})_E](s_2',h_2')\big)\ \vee$
$\big([\![B]\!]_{s_1} = \mathsf{false}\ \wedge\ c_2,(s_1,h_1) \rightsquigarrow^i_{(p,\eta)} (s_1',h_1')\ \wedge\ (s_1',h_1')[(R*\mathsf{Id})_E](s_2',h_2')\big)$

$\Longrightarrow$ (by the semantics of the if-then-else statement)

$\exists(s_1',h_1').\ \big(c,(s_1,h_1) \rightsquigarrow^i_{(p,\eta)} (s_1',h_1')\ \wedge\ (s_1',h_1')[(R*\mathsf{Id})_E](s_2',h_2')\big).$

Let $c \equiv$ **while** B **do** c_1. In this case we do an inner induction on the length of the derivation for the **while**-loop. Suppose the theorem is true for all the derivations of length n or less. Consider two starting normal or exception states d_1, d_2 such that $d_1[(R*\mathsf{Id})_E]d_2$. Then, because of the definition of $(R*\mathsf{Id})_E$, either both d_1 and d_2 are normal states in Σ, or they are exception states in E. When d_1 and d_2 are exception states, they are the only outputs of the abstract and concrete commands at d_1 and d_2, respectively. This theorem follows from this fact. Now suppose the starting states are not exception states. If the derivation is of length 0, then $[\![B]\!]_{s_2} = \mathsf{false}$. Hence, $[\![B]\!]_{s_1} = \mathsf{false}$, because $(s_1,h_1)[R*\mathsf{Id}](s_2,h_2)$ and B does not contain any module variables. In the semantics instrumented by (p,η) and the one instrumented by (q,μ), the initial states are unchanged by the execution, and so the theorem holds in this case. For the inductive case where the length of the derivation is $n+1$, we note that the rule below must hold

$$\frac{[\![B]\!]_s = \mathsf{true} \qquad (c_1; \textbf{while } B \textbf{ do } c_1),(s_2,h_2) \rightsquigarrow^i_{(q,\mu)} K}{(\textbf{while } B \textbf{ do } c),(s_2,h_2) \rightsquigarrow^i_{(q,\mu)} K}$$

where K is wrong, av or a state (s_2',h_2'). Now, we apply the outer induction hypothesis to c_1 and the inner induction hypothesis to **while** B **do** c_1. The conclusions of these applications let us use Lemma 5 to prove the theorem. □

Remark on Incompleteness and Backward Simulation. Here, we have considered only forward simulation. In the literature, backward simulations are considered as well. In [HHS86] backward simulations (called up-simulations there) and forward simulations are used together to prove a completeness result. We do not follow that approach here, for the very simple reason that we know that we are in a position of being incomplete, because of our reliance on growing relations for soundness.

Furthermore, our method proves refinement, but not equivalence, between modules. other work, such as [] approaches equivalence, but with many more restrictions on programs than here.

There are two open problems raised by this discussion.

- Open Problem 1. Find a notion of simulation that allows us to prove equivalence between programs (or program components) that use pointers, without undue restrictions on the programs [BN05a].
- Open Problem 2. Obtain a completeness theorem for refinement of pointer programs.

10. Interference-free Lifting and Standard Semantics

The lifting theorem in the previous section says that if we have a simulation between an abstract and concrete module, then this lifts to all client programs, *but for the instrumented semantics only*. The discussion in Section 2 provides a counterexample to the lifting theorem for the standard semantics. But, the reader might react: we have proven the theorem we want, but for a non-standard semantics; how can we interpret that? The main point is that when the client program does not interfere with module internals, the instrumented theorem tells us a standard fact about lifting a simulation.

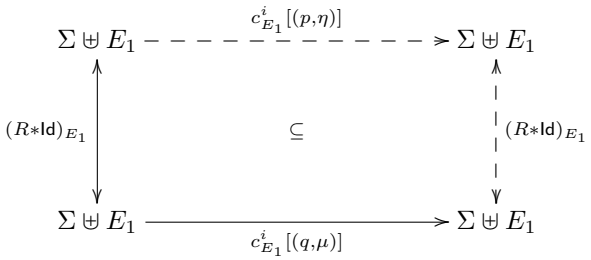
In this section we nail these intuitions down by connecting the standard and instrumented semantics.

In preparation for the second main result of the paper we say when a client context is interference-free.

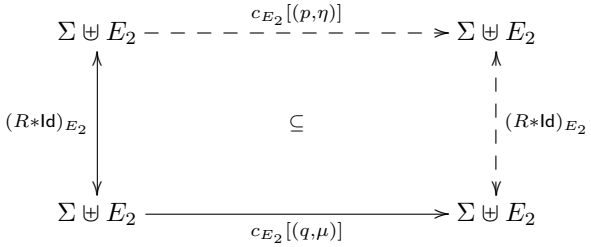
Definition 7 (Interference Freedom). Let (p, η) be a fixed module. We say that $c[(p, \eta)]$ is **interference-free** if it does not access violate when run in a start state in which p holds (at some substate):

$$\forall (s, h) \in p * \mathsf{true}.\ \ c, (s, h)\ \not\rightsquigarrow^{i}_{(p,\eta)}\ \mathsf{av}.$$

Lemma 6 (Linking lemma). Let (p, η) and (q, μ) be two modules such that $\mathsf{dom}(\mu) = \mathsf{dom}(\eta)$ and let $R * \mathsf{Id}$ be a growing relation between them. Also let sets E_1 and E_2 denote the following exception state sets: $E_1 = \{\mathsf{wrong}, \mathsf{av}\}$ and $E_2 = \{\mathsf{wrong}\}$. If the following diagram holds for all contexts $c[\cdot]$



then for all contexts $c[\cdot]$ for which $c[(p, \eta)]$ is interference-free, the following diagram also holds.



Proof. Pick a context $c[\cdot]$ for which $c[(p, \eta)]$ is interference-free. Let d_1, d_2 and d_2' be such that $d_1[(R*\mathsf{Id})_{E_2}]d_2$ and $d_2\big[c_{E_2}[(q, \mu)]\big]d_2'$. We should show the existence of d_1' satisfying

$$d_1\big[c_{E_2}[(p, \eta)]\big]d_1' \quad \text{and} \quad d_1'[(R * \mathsf{Id})_{E_2}]d_2'.$$

We will prove this lemma by case analysis on whether $d_1\big[c_{E_2}[(p, \eta)]\big]\mathsf{wrong}$ or not.

Suppose that

$$d_1\ \big[c_{E_2}[(p, \eta)]\big]\ \mathsf{wrong}.$$

Then, the required d_1' is wrong. To see this, note that d_2' is in $\Sigma \cup \{\mathsf{wrong}\}$ and the relation $(R * \mathsf{Id})_{E_2}$ relates wrong to every element in $\Sigma \cup \{\mathsf{wrong}\}$. Thus,

$$\mathsf{wrong}[(R * \mathsf{Id})_{E_2}]d_2'.$$

This, together with our supposition $d_1[c_{E_2}[(p, \eta)]]\mathsf{wrong}$, shows that wrong is the element that we are looking for.

Now, assume the other case that d_1 is not related to wrong by $c_{E_2}[(p, \eta)]$. We will prove this case in two steps. Firstly, we will show that d_1 is not related to av nor wrong by $c^i_{E_1}[(p, \eta)]$. Note that in this case, by our assumption,

$$\neg\Big(d_1\ [c^i_{E_1}[(p, \eta)]]\ \mathsf{wrong}\Big), \tag{13}$$

because our instrumented semantics and normal semantics coincide for non-av outputs. By the definition of $c_{E_2}[(p, \eta)]$, the assumption also implies that d_1 cannot be wrong. From this, we will derive the remaining claim in this first step:

$$\neg\Big(d_1\ [c^i_{E_1}[(p, \eta)]]\ \mathsf{av}\Big). \tag{14}$$

Since $d_1[(R * \mathsf{Id})_{E_2}]d_2$ and d_1 is not wrong, we must have that

$$d_1[R * \mathsf{Id}]d_2.$$

Because the domain of $R * \mathsf{Id}$ is $p * \mathsf{true}$, this relationship implies that d_1 is a normal state in $p * \mathsf{true}$. Now, using the interference freedom of $c[(p, \eta)]$, we can conclude the property 14 above. Secondly, we prove the lemma. By the choice of d_1, d_2 and d_2', we have that

$$d_1[(R * \mathsf{Id})_{E_2}]d_2 \qquad \text{and} \qquad d_2\big[c_{E_2}[(q, \mu)]\big]d_2'.$$

Note that

$$(R * \mathsf{Id})_{E_2} \subseteq (R * \mathsf{Id})_{E_1} \qquad \text{and} \qquad c_{E_2}[(q, \mu)] \subseteq c^i_{E_1}[(q, \mu)].$$

Thus, we can apply the assumed upper diagram of this lemma, and get d_1' such that

$$d_1\big[c^i_{E_1}[(p, \eta)]\big]d_1' \qquad \text{and} \qquad d_1'[(R * \mathsf{Id})_{E_1}]d_2'.$$

By what we have just shown in the first step, d_1' cannot be av or wrong. Thus, both d_1' and d_2' are normal states in Σ. This gives the desired

$$d_1\big[c_{E_2}[(p, \eta)]\big]d_1' \qquad \text{and} \qquad d_1'[(R * \mathsf{Id})_{E_2}]d_2',$$

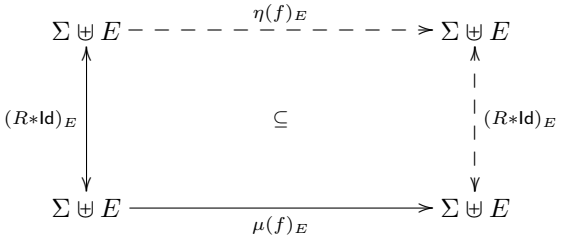
because the instrumented semantics and the normal semantics coincide for non-av outputs and

$$(R * \mathsf{Id})_{E_1} \cap (\Sigma \times \Sigma) \quad = \quad (R * \mathsf{Id})_{E_2} \cap (\Sigma \times \Sigma).$$

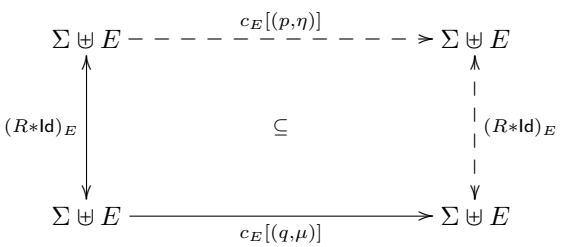
We have just shown that d_1' is the element that the lower diagram of this lemma requires. □

Then, lifting holds for the standard semantics, *as long as the abstract client is interference-free*. We emphasize here that there is no need to explicitly check for interference freedom in the client module: that is taken care of by simulation. There is a useful methodological point here. When we have proven a client program with respect to an abstract module, and then we provide a refinement, there is no need to re-check the client for interference-freedom.

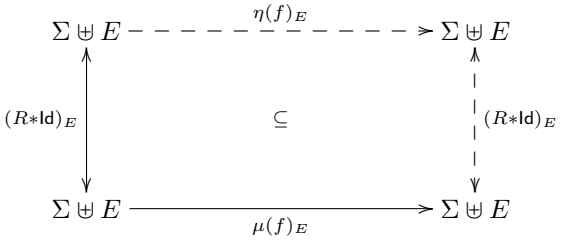
Theorem 2 (Interference-free Lifting Theorem). Let (p, η) and (q, μ) be two modules such that $\mathsf{dom}(\mu) = \mathsf{dom}(\eta)$ and let $R * \mathsf{Id}$ be a growing relation between them. Also, let E denote the singleton set $\{\mathsf{wrong}\}$. If for all $f \in \mathsf{dom}(\mu)$ the diagram



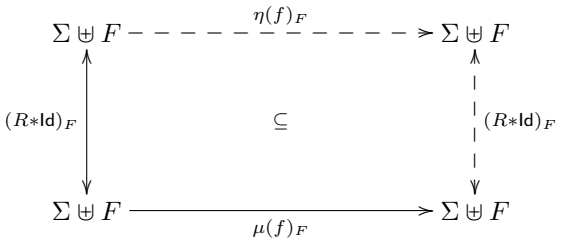
holds, then for all contexts $c[\cdot]$ for which $c[(p, \eta)]$ is interference-free, the following diagram holds, too.



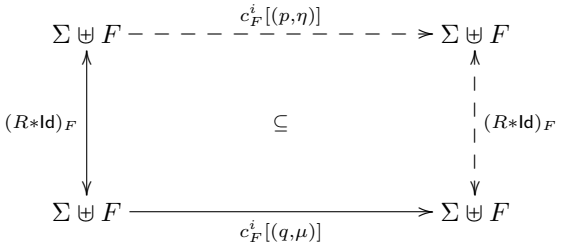
Proof. Suppose that for all $f \in \mathsf{dom}(\mu)$ the diagram



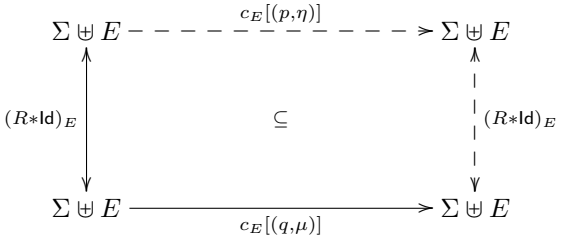
holds. Then, it is easy to see that, for all $f \in \mathsf{dom}(\mu)$, the diagram



also holds, where $F = \{\mathsf{wrong}, \mathsf{av}\}$. The Instrumented Lifting Theorem then implies that for all contexts $c[\cdot]$, the following diagram holds, too.



Now, the conditions of the Linking lemma are satisfied, and so we can apply it. It follows that the diagram



holds, which is exactly what we wanted to prove. □

An interesting special case of this result is when the representation invariant p in an abstract module is emp. That is, the abstract module does not contain any heap resource. In that case, interference freedom is automatic, and so lifting for the standard semantics holds. This special case often occurs, typically, whenever we start with a "mathematical" description of a data type, and then refine it to a linked representation. For example, we can specify a queue module abstractly using a variable Q ranging over sequences, and hold the heap empty. We provide a refinement to a linked representation. Then, the Interference-Free Lifting Theorem applies to all clients.

Note. In Mijajlović's thesis [Mij07] a notion of a separation context was used to achieve the effect of the Interference-Free Lifting Theorem. Namely, in the thesis separation context was a central notion around which the whole theory was built, while here a different perspective is given.

11. Refinement

As in [HHS86], we can define the notion of refinement of modules independently of simulation. Refinement is a relation defined in terms of observations made on complete programs, without reference to simulations. Then, simulation is viewed as a proof technique for establishing refinement. In this section we tie our work on simulation together with this kind of view on refinement. To do so, we must include initialization and finalization operations for a module, which were left out up to now, in order to slot the modules into common contexts where programs' values can be observed.

This section is technically straightforward given our previous work, so we proceed somewhat less formally, and without providing proofs.

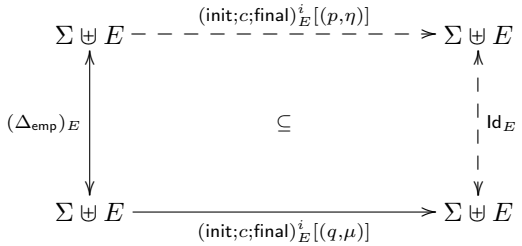
First, we suppose that any module comes along with additional operations, init and final. An interpretation of init and final in a module (p, η) must satisfy the following two properties.

1. $\eta(\mathsf{init}) \subseteq \{\mathsf{emp}\} - \{p\}$
2. $\eta(\mathsf{final}) \subseteq \{p\} - \{\mathsf{emp}\}$

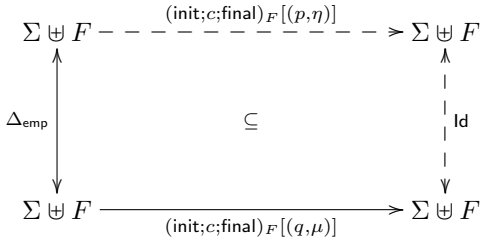
These conditions express the idea that the initialization creates the resource p, and the finalization annihilates it. Our contexts c now range over commands built from client operations and module operations other than initialization and finalization.

We define two notions of refinement, one of which involves our "trumping simulations" R_E, where $E = \{\mathsf{wrong}, \mathsf{av}\}$, (Section 8) and the other R_F, where $F = \{\mathsf{wrong}\}$, of which is the traditional notion of refinement.

Definition 8 (Refinement). 1. Module (q, μ) is a **trumped refinement** of (p, η) if for all contexts c,



2. Module (q, μ) is a **trad refinement** of (p, η) if for all contexts c,



Next, we extend the notion of simulation between modules to account for initialization and finalization.

Definition 9. Module (q, μ) simulates (p, η) if

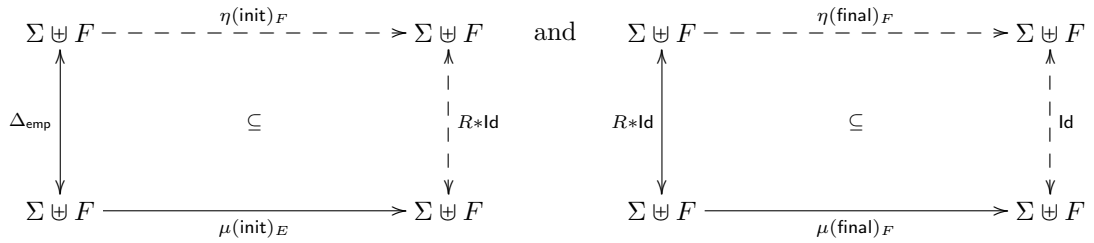


Table 7. Queue Module Specification

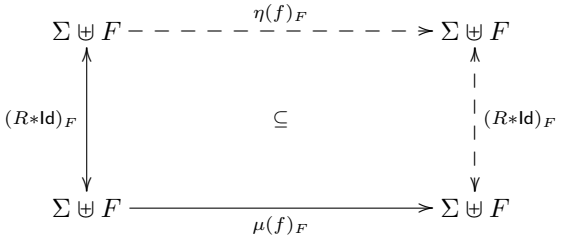
Concrete Queue module: $(\exists\alpha.\ \mathsf{listseg}(x,\alpha,y) * y \mapsto -,-,\ \mu_c)$, where

$$\begin{array}{l}\mu_c(\mathsf{enq}()) = \{\mathsf{listseg}(x,\alpha,y)\ \wedge\ z=n\} - \{\mathsf{listseg}(x,\alpha\cdot\langle n\rangle,y\}[y]\\ \mu_c(\mathsf{deq}()) = \{\mathsf{listseg}(x,\langle n\rangle\cdot\alpha,y)\} - \{\mathsf{listseg}(x,\alpha,y)\ \wedge\ z=n\}[x,z]\end{array}$$

Abstract Queue module: $(\exists\alpha.\ Q=\alpha\ \wedge\ \mathsf{emp},\ \mu_a)$, where

$$\begin{array}{l}\mu_a(\mathsf{enq}()) = \{Q=\alpha\ \wedge\ z=n\ \wedge\ \mathsf{emp})\} - \{Q=\alpha\cdot\langle n\rangle\ \wedge\ \mathsf{emp}\}[Q]\\ \mu_a(\mathsf{deq}()) = \{Q=\langle m\rangle\cdot\alpha\ \wedge\ \mathsf{emp}\} - \{Q=\alpha\ \wedge\ z=m\ \wedge\ \mathsf{emp}\}[Q,z].\end{array}$$

hold, and for all other module operations f,



holds.

Theorem 3 (Refinement Theorem). Suppose that module (q,μ) simulates (p,η). Then (q,μ) is a trumped refinement of (p,η). Furthermore, if the abstract module is heap-free, so that p is emp, then (q,μ) is a trad refinement of (p,η).

12. A Larger Example

We provide a larger example to tie the pieces of the paper together. Our intention is to illustrate the "programming methodology" aspect that our work lends to separation logic. We show how we can refine several modules independently, despite the fact that they both share the same memory-array (the heap). This provides for the heap the same kind of decomposition that standard refinement reasoning provides by choosing different variables (or different arrays) in the representation of different modules [Jon80, GM91].

For our illustration we consider a combination of a memory manager and a queue. These were treated in a post-hoc verification in [OYR09]. The theory there allowed the modules to be verified in a modular fashion. Our work goes beyond that in supporting a developmental treatment, by refinement. We will display the representation invariants and simulation relations for the example, but will not carry out detailed mathematical proofs that operations preserve the simulation relations.

Let us first consider the Queue module. The abstract and concrete operations are specified using greatest relations. These specifications together with resource invariants for each module are given in Table 7. Here, α, n and m are ghost variables, x and y are concrete module variables, Q is an abstract module variable, and z is a client variable. In the abstract module the queue is represented by a mathematical sequence of integers, and operations $\mathsf{enq}()$ and $\mathsf{deq}()$ are defined as appending an element to the end of the sequence and obtaining the head element of the sequence, respectively. On the other hand, the queue is represented as a linked list segment in the concrete module. In order to specify the invariant of the concrete module, we take the listseg predicate from [OYR09].

$$\begin{array}{rcl}\mathsf{listseg}(x,\alpha,y) & \stackrel{def}{=} & \text{if } x=y \text{ then } (\alpha=[]\ \wedge\ \mathsf{emp})\\ & & \text{else } (\exists v,z,\alpha'.\ (\alpha=\langle v\rangle\cdot\alpha'\ \ \wedge\ x\mapsto z,v) * \mathsf{listseg}(z,\alpha',y))\end{array}$$

As the module invariant of the concrete module indicates, the concrete module keeps a sentinel at the end of its internal list. The sentinel is used to reserve space for a value to be enqued, and in consistent state does not hold any values.

The definitions of the abstract and concrete queue modules are in Table 7.

Table 8. Queue module, in programming notation

Abstract Representation	**Concrete Representation**
FiniteSequence Q;	slink $*$x, $*$y;
Rep Invariant : $\exists\alpha.\ \mathsf{Q} = \alpha\ \wedge$ emp	Rep Invariant : $\exists\alpha.$ listseg(x, α, y) $*$ y $\mapsto -, -$

void enq(**int** n){
Q = Q $\cdot$ ⟨n⟩; }
	void enq(**int** n){
	slink $*$t;
	t = talloc();
	y $\rightarrow$ nxt = t; y $\rightarrow$ data = n; y = t; }

int deq() // Pre : Q $\neq$ []
{**int** res;
res := car(Q); Q = cdr(Q)
return res; }
	int deq() // Pre : x $\neq$ y
	{slink $*$t; **int** res;
	res = x $\rightarrow$ data; t = x; x = x $\rightarrow$ nxt; tfree(t);
	return res; }

We now give the relation that connects corresponding states of the abstract and concrete modules.

$$R = \{((s_1, h_1), (s_2, h_2)) \mid \exists\alpha.\ ((s_1, h_1) \models Q = \alpha \wedge \mathsf{emp})\ \text{ and }\ (s_2, h_2) \models \mathsf{listseg}(x, \alpha, y) * y \mapsto -, -\}$$

The relation connects the resource invariants of the modules by requiring that an abstract sequence and a list representing the queue contain exactly the same elements and in the same order. More succinctly, they need to represent the same queue. Now, once it is proved that the pairs of corresponding operations preserve simulation relation R, the lifting theorem ensures that the concrete Queue module implementation can be used instead of the abstract one in any client program of the Queue module.

Table 7 gives the specification of the queue in mathematics, without mentioning programs. However, we can just as well write programs that satisfy the greatest relations: these are given in Table 8. Now, if we regard the code in the concrete module in Table 8 as client code for another module, the memory manager module, we can go ahead and independently refine it. For, the reader will have noticed that the concrete implementation of the queue operations assumes calls to the operations of the Toy Memory Manager module, namely, talloc() and tfree().

The top step of this part of our refinement has to be at a more abstract level than our previous definitions of the memory manager. This most abstract, "Magical" memory manager does not keep any resources. Instead, cells materialize (as if out of nowhere) when talloc() is called and they disappear when tfree() is called. The Magical manager has resource invariant emp, and is defined in Table 9 which, for comparison as we define our simulation relations, also repeats the definitions of the abstract and concrete modules from before (Table 5).

In terms of code, you could think of talloc() and tfree() simply calling the C built-in functions `malloc`() and free(), rather than maintaining internal data structures.

```
Rep Invariant : emp

slink *talloc(){
  return(slink *)malloc(sizeof(slink)); }

void tfree(x) // Pre : x ↦ −, −
  {free(x); }
```

This code is given for illustration only; the official definition is in Table 9.

The Magical memory manager is the one which shows that the code in Table 8 satisfies the specifications in Table 7. Now, we can refine the memory manager without again revisiting this question. The Refinement Theorem (and the lifting theorems) ensure that we automatically obtain a more refined queue implementation.

Now, we can view the queue operations, enq() and deq() as the clients of the Toy Memory Manager and after providing adequate simulation relations and proving that the implementations given in Table 1 are refinements of the Magical memory manager module, we can use any of those two implementations instead of the Magical one.

Table 9. Three Memory Manager Modules

Magical toy memory manager module $(\mathsf{emp},\ \eta_m)$, where

$$\begin{array}{rl} \eta_m(\mathsf{talloc}()) : & \{\mathsf{emp}\} - \{x \mapsto -, -\}[x] \\ \eta_m(\mathsf{tfree}()) : & \{x \mapsto -, -\} - \{\mathsf{emp}\}[]. \end{array}$$

Concrete toy memory manager module: $(\exists\alpha.\ \mathsf{list}(\alpha, fl), \eta_c)$, where

$$\begin{array}{rl} \eta_c(\mathsf{talloc}()) = & \{\mathsf{list}(a\cdot\alpha, fl)\} - \{\mathsf{list}(\alpha, fl) * a \mapsto -, - \ \wedge\ x = a\}[x, fl] \\ & \{\mathsf{list}(\epsilon, fl)\} - \{\mathsf{list}(\epsilon, fl) * x \mapsto -, -\}[x, fl] \\ \eta_c(\mathsf{tfree}()) = & \{\mathsf{list}(\alpha, fl) * x \mapsto -, - \ \wedge\ x = a\} - \{\mathsf{list}(a\cdot\alpha, fl)\}[fl] \end{array}$$

Abstract toy memory manager module: $(\exists\alpha.\ \mathsf{Set}(\alpha, S), \eta_a)$, where

$$\begin{array}{rl} \eta_a(\mathsf{talloc}()) = & \{\mathsf{Set}(a\cdot\alpha, S)\} - \{\mathsf{Set}(\alpha, S) * a \mapsto -, - \ \wedge\ x = a\}[x, S] \\ & \{\mathsf{Set}(\epsilon, S)\} - \{\mathsf{Set}(\epsilon, S) * x \mapsto -, -\}[x, S] \\ \eta_a(\mathsf{tfree}()) = & \{\mathsf{Set}(\alpha, S) * x \mapsto -, - \ \wedge\ x = a\} - \{\mathsf{Set}(a\cdot\alpha, S\}[S]. \end{array}$$

We consider successive refinements, namely, the refinement between the Magical and the set-based implementation, as well as the refinement between the set-based and the linked-list-based implementation of the Toy Memory Manager module. The corresponding simulation relations are:

$$\begin{array}{rcl} R_{12} & = & \{((s_1,h_1),(s_2,h_2)) \mid (s_1.h_1) \models \mathsf{emp} \ \text{ and } \ (s_2,h_2) \models \exists\alpha.\ \mathsf{Set}(\alpha, S)\} \\ R_{23} & = & \{((s_2,h_2),(s_3,h_3)) \mid \exists\alpha.\ (s_2,h_2) \models \mathsf{Set}(\alpha, S) \ \text{ and } \ (s_3,h_3) \models \mathsf{list}(\alpha, fl)\} \end{array}$$

We can summarize the result obtained as follows in terms of programs. The "most abstract" queue module is the one that does not mention the memory manager at all, as in Table 8. The "most concrete" queue corresponds to the concrete representation in Table 8, paired with the concrete representation of the memory manager in Table 1 from Section 2. We have connected these most abstract and most concrete representations by a series of three simulation steps, one on the queue itself and two on the manager. Since the most abstract representation is heap-free, we may conclude by the Refinement Theorem (Theorem 3) that the most concrete representation is a refinement, in the traditional sense, of the most abstract.

Furthermore, our simulation arguments concerning the queue and manager were done independently. This is despite the fact that the queue and the manager share resources; they might both use the *same* cell in their internal representations. To see this, a client program

$\mathsf{enq}();\ \mathsf{deq}()$

can cause a cell to transfer from the memory manager to the queue, and then back to the memory manager. But, although both the manager and the queue might use the same cell, they do so at different times. There is a *dynamic separation* between the resources of the queue and the manager where, at any time, a heap cell can belong to only one or the other. It is this fact that has allowed us to refine the queue and the manager independently. An illustration of this dynamic division is given in Figure 1.

13. Related Work

Some related work was mentioned in Section 3, when we described our point of departure. This section discusses other and more recent related work.

Butler and Back *et al.* have considered pointers from the point of view of formalisms for refinement [But99, BFP03]. They are concerned with the problem of how to go about establishing a simulation relation, while we leave the method of doing so completely unspecified. On the other hand, we are concerned with lifting simulations from modules to clients and (implicitly) refinement of multiple modules into the same memory-array, while they do not consider refinements into the same array. So, their work is complementary to that here. Indeed, we could use their techniques when reasoning inside a single module.

One of us has considered using separation logic to do simulation proofs [Yan07]. It is shown how the Schorr-Waite graph-marking algorithm can be connected by a simulation relation to a depth-first traversal using an explicit stack; the pointer-reversal trick simulates the stack. Again, that work is complementary to that here; we could use those techniques as a means to help us establish that simulation results do hold.

Banerjee and Naumann's theory of confinement is the first work that made headway on the soundness

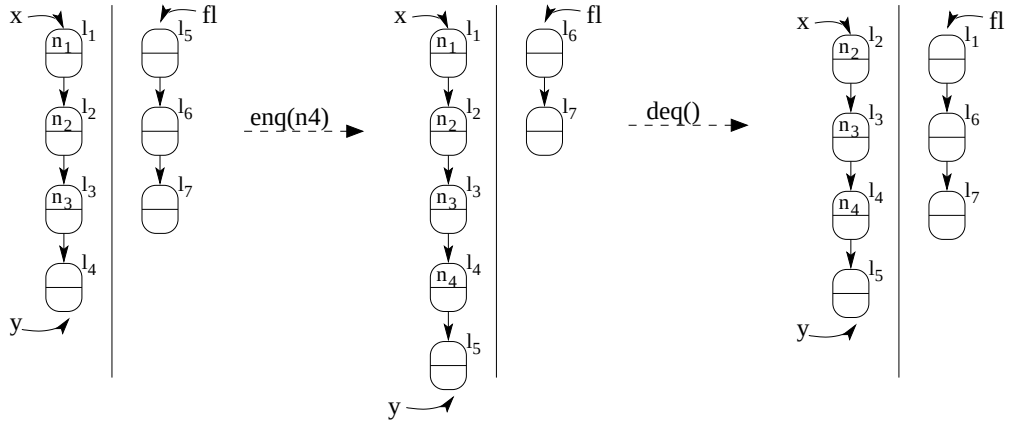


Fig. 1. Dynamic heap division between the Queue and Memory Manager modules

problems that pointers cause for simulation reasoning [BN05a]. The research here was partly inspired by their work, both the successful steps it made, but also the limitations it had. Banerjee and Naumann's is perhaps the most advanced (theoretically) of the work that "blames the modules", and it rules out many interesting programs:[4] we felt that the restrictions it used were not necessary to sound simulation reasoning.

Subsequent to the first version of this work [MTSO04], Banerjee and Naumann went on to lift many of their restrictions using what appears to be a similar (in a general sense) point of departure to ours: they blame the clients as well [BN05b]. Their approach follows on from the Boogie methodology [BDF$^+$04], and logic is used to rule out bad clients (as well as to constrain modules). See the survey articles [LML07, Nau07] for an account of much recent work in this line.

It is difficult to precisely compare [BN05b] against the work here. They take an almost maximalist approach, where they try to handle as many programming features of Java as possible in order to show the realism of their theory. We take a minimalist approach, where we exclude as many features as possible to get to a small language that still causes us theoretical problems. Their theory works for a single language (which is Java-like), while ours is parametrized by client as well as module operations satisfying abstract healthiness conditions. Furthermore, one of our example models, the RAM, corresponds more directly to what low-level assembly-language programs can see than object-oriented programs. The intuition of separation is present in both works; here it is a primitive while in [BN05b] it is encoded in an ownership hierarchy.[5]

Banerjee and Naumann have repeatedly criticized this work and that in [OYR09] for dealing only with single-instance modules and for not dealing with callbacks. Of course, this is partly due to the minimalist approach we have taken, which we do not apologize for. We do wonder if taking a higher-order separation logic [PB05, BBTS07] (rather than [OYR09]) as a starting point might deal with some of these problems smoothly. Parkinson has convincingly argued (see [Par07]) that the combination of separation and higher-order logic gives a simpler account of abstraction for classes and callbacks than do the approaches in, e.g., [Nau07]. Lifting these insights about separation and abstraction to refinement and simulation is a possible (non-trivial) direction for future work.

We would like to take this opportunity to offer a long overdue rejoinder to Banerjee and Naumann. Our theory works without difficulty in the presence of pointer arithmetic, and applies to C-like and assembly programs, while their theory (at least how it is presented) assumes the high-level abstractions of an object-oriented language. It would be most unfortunate if refinement could only be applied to higher-level programs, which are realized by a closure abstraction, and which rely on a garbage collector. Surely, the intuition of refinement does not depend on such abstractions. As well as for "clean" programming languages, the theory of refinement should be applicable to the low-level programs in operating systems, network code, and device drivers that power the world's computing infrastructure. This work is one attempt to make it so.

[4] We stress that their theory accepts many programs as well, perhaps the majority if one excludes programs in low-level languages.

[5] It seems highly unlikely that the hierarchy, which is restrictive, is in any way necessary to their work, especially judging from recent developments such as [BNR08].

All jousting aside, there is a significant technical point of divergence. Banerjee and Naumann (and Benton [Ben06]) suggest that our problems that lead to the need for growing relations simply disappear if one disallows the nondeterministic semantics of the memory allocator (as we allow). Try as we might, we have been unable to see that their suggestion can be upheld, for the following reason. A genuine theory of refinement should not apply only to a specific collection of programs, with results proven by induction on syntax. Rather, it should apply to an abstract collection of potential program meanings, including meanings determined by "specification statements" (pre/post pairs). It is difficult to rule out nondeterminism from the specification statements. For, even if the allocator in a given language is deterministic, one might write one's own allocator which we, as the specifier, intentionally give a nondeterministic specification. In frequent arguments about this issue over the years in London, we have called this module "Peter's memory manager"; it has pmalloc and pfree operations where pmalloc is a nondeterministic selection from a set of locations. Even if you give us a garbage collected language, Peter will produce his own memory manager module with pmalloc and pfree.[6] For this contention of Banerjee, Naumann and Benton to be upheld, it must work for pmalloc and pfree, and not only the language's built-in manager; else, the solution will fly in the face of modularity. Furthermore, it must work for specification statements that Peter, or anyone else, might write, and not just for programs. (We extend this as a friendly challenge to our colleagues, which we hope could be answered in a minimalist way, without reference to objects or ownership.)

Other work that takes on the problems caused by pointers for simulation reasoning include [RY03, BL05, BY07]. Along with the work here and that of [BN05a, BN05b], progress on these problems have been made in recent years but, in general, no final solution can be claimed.

14. Conclusions

In this paper we have described a sound theory of data refinement in the presence of pointers. It has two components: a method of "blaming the client", using instrumented semantics, and a restricted form of simulation relations, growing relations. A simulation given by a growing relation can be lifted to all the clients that don't interfere with an abstract module. Our theory is such that once non-interference is established with respect to the abstract module, we don't have to make a check for a concrete module obtained by refinement.

We admit some discomfort with the need to restrict to growing relations. In separate work this restriction is avoided, at some cost in complexity [MY05]. On the other hand, we have no discomfort whatsoever with our decision to blame the clients. It seems impossible to have a usable notion of data refinement, that covers real-world systems programs, in which refinement can automatically be lifted to *all* program contexts. Freedom from interference is too difficult to be absolutely guaranteed within a module, independently of the clients, for advanced programming patterns.

We are not alone in the contentions made in this paper. Other authors (see Section 13) have come to what appears to be the same conclusion concerning blaming the clients, even if we and they use different technical machinery to express the ideas. Fundamentally, because it concerns aliased pointers, freedom from interference is extremely difficult to protect against using programming language restrictions, and too expensive to protect against with runtime checking. It is better to say that *if* there is no interference *then* refinement reasoning is sound, rather than to say that it is unconditionally sound.

Acknowledgments. We are grateful to Richard Bornat for discussions concerning separation contexts at the beginning of this work, and to Josh Berdine, Cristiano Calcagno and other members of the East London Massive for numerous dissections of both conceptually and technically incorrect solutions to the intricate problems surrounding simulation and allocation. We have benefitted from a long-running conversation with Anindya Banerjee and David Naumann on pointers and simulation-based proof techniques.

The research of the London authors was supported by the EPSRC. O'Hearn acknowledges the support of a Royal Society Wolfson Research Merit Award. At the time of conducting this research, Torp-Smith was affiliated with the IT University of Copenhagen and was partially sponsored by the Danish Technical Research Council Grant 56-00-0309.

[6] Just as how in reality Java programmers sometimes program their own resource management strategies, which are not taken care of by the garbage collector.

Some of the material in this paper was published in preliminary form in the conference papers [MTSO04, MY05].

References

[Bac78] R. J. Back. On the correctness of refinement steps in program development. Technical Report A-1978-4, Department of Computer Science, University of Helsinki, 1978.

[Bac80] R. J. Back. Correctness preserving program refinements: proof theory and applications. Volume 131 of Mathematical Centre Tracts, Mathematisch Centrum, Amtserdam, 1980.

[BBTS07] B. Biering, L. Birkedal, and N. Torp-Smith. BI-hyperdoctrines, higher-order separation logic, and abstraction. *ACM Trans. Programming Languages and Systems*, 29(5), 2007.

[BDF+04] M. Barnett, R. DeLine, M. Fähndrich, K. Rustan M. Leino, and Wolfram Schulte. Verification of object-oriented programs with invariants. *JOT*, 3(6), 2004.

[Ben06] N. Benton. Abstracting Allocation: The New new thing. In *Proceedings of Computer Science Logic (CSL '06)*, volume 4207 of *LNCS*, 2006.

[BFP03] R. J. Back, X. Fan, and V. Preoteasa. Reasoning about pointers in refinement calculus. In *Proceedings of the Tenth Asia-Pacific Software Engineering Conference (APSEC'03)*, 2003.

[BL05] N. Benton and B. Leperchley. Relational reasoning in a nominal semantics for storage. In *In* 7th TLCA*, LNCS 3641*, pages 86–101, 2005.

[BN05a] A. Banerjee and D. Naumann. Ownership confinement ensures representation independence in object-oriented programs. *Journal of the ACM*, 52(6):894 – 960, November 2005.

[BN05b] A. Banerjee and D. Naumann. State based ownership, reentrance and encapsulation. In *Proceedings of the Nineteenth European Conference on Object-oriented Programming (ECOOP)*, volume 3586 of *LNCS*, pages 387 – 411. Springer-Verlag, July 2005.

[BNR08] A. Banerjee, D. Naumann, and S. Rosenberg. Regional logic for local reasoning about global invariants. In *Proceedings of the 22nd European Conference on Object-oriented Programming (ECOOP)*, volume 5142 of *LNCS*, pages 387–411. Springer-Verlag, July 2008.

[Bor00] R. Bornat. Proving pointer programs in Hoare logic. In *Mathematics of Program Construction*, 2000.

[Bro07] S. D. Brookes. A semantics of concurrent separation logic. *Theoretical Computer Science*, 375(1-3):227–270, 2007. (Preliminary version appeared in CONCUR'04, LNCS 3170, pp16-34).

[But99] M. Butler. Calculational derivation of pointer algorithms from tree operations. *Science of Computer Programming*, 33:221–260, 1999.

[BY07] L. Birkedal and H. Yang. Relational parametricity and separation logic. In *10th FOSSACS*, 2007.

[CNP01] D. G. Clarke, J Noble, and J. M. Potter. Simple ownership types for object containment. In *Proc. European Conference on Object-Oriented Programming*, June 2001.

[dRE98] Willem-Paul de Roever and Kai Engelhardt. *Data Refinement: Model-Oriented Proof Methods and their Comparison.* Number 47 in Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, Cambridge, UK, November 1998.

[FOB05] James C. Foster, Vitaly Osipov, and Nish Bhalla. *Buffer overflow attacks: Detect, Exploit, Prevent.* Syngress Publishing, Inc., 2005.

[GM91] P. H. Gardiner and C. C. Morgan. Data refinement of predicate transformers. *Theoretical Computer Science*, 87:143–162, 1991.

[HH90] CAR Hoare and J. He. Data refinement in a categorical setting. Technical Report PRG-90, Oxford University Computing Laboratory, November 1990.

[HHS86] J. He, C. A. R. Hoare, and J. W. Sanders. Data refinement refined (resume). In B. Robinet and R. Wilhelm, editors, *ESOP 86, European Symposium on Programming*, volume 213 of *Lecture Notes in Computer Science*, pages 187 – 196. Springer Verlag, 1986.

[Hoa72] C. A. R. Hoare. Proof of correctness of data representations. *Acta Informatica*, 1:271 – 281, 1972.

[Hog91] J. Hogg. Islands: Aliasing protection in object-oriented languages. In *OOPSLA'91*, 1991.

[IO01] S. Ishtiaq and P. W. O'Hearn. BI as an assertion language for mutable data structures. In *Principles of Programming Languages*, volume 28, London, 2001. ACM - SIGPLAN.

[Jon80] C. B. Jones. *Software development: a rigorous approach.* Prentice-Hall, 1980.

[jp01] Advanced Doug Lea's malloc exploits. Internet page, 2001. Available at `http://doc.bughunter.net/buffer-overflow/advanced-malloc-exploits.html`.

[Kae01] Michel "MaXX" Kaempf. Smashing the heap for fun and profit. Internet page, 2001. Available at `http://doc.bughunter.net/buffer-overflow/heap-corruption.html#gnu_malloc`.

[Knu73] D.E. Knuth. *The Art of Computer Programming, Volume I: Fundamental Algorithms.* Addison-Wesley, 1973. 2nd Edition.

[KR88] B. W. Kernighan and D. M. Ritchie. *The C Programming Language.* Prentice Hall, New Jersey, 1988. 2nd Edition.

[Lea01] Dougl Lea. A memory allocator. Internet page, 2001. Available at `http://g.oswego.edu/dl/html/malloc.html`.

[LML07] G. Leavens, P. Müllen, and K.R.M. Leino. Specification and verification challenges for sequential object-oriented programs. *Formal Asp. Comput. 19(2): 159-189*, 2007.

[Mij07] I. Mijajlović. *Separation and Data refinement.* PhD thesis, Queen Mary, University of London, 2007.

[MRG88] C. Morgan, K. Robinson, and P. Gardiner. On the refinement calculus. Technical Report PRG-70, Oxford University Computing Laboratory, October 1988.

[MTSO04] I. Mijajlović, N. Torp-Smith, and P. O'Hearn. Refinement and separation contexts. In K. Lodaya and M. Mahajan, editors, *FSTTCS*, volume LNCS 3328, pages 421–433, 2004.

[MY05] I. Mijajlović and H. Yang. Data refinement with low-level pointer operations. In Kwangkeun Yi, editor, *Programming Languages and Systems*, volume LNCS 3780, pages 19–36, November 2005.

[Nau07] D. Naumann. On assertion-based encapsulation for object invariants and simulations. *Formal Asp. Comput. 19(2): 205-224*, 2007.

[O'H07] P. W. O'Hearn. Resources, concurrency and local reasoning. *Theoretical Computer Science*, 375(1-3):271–307, 2007. (Preliminary version appeared in CONCUR'04, LNCS 3170, pp49-67).

[OP99] P. O'Hearn and D. J. Pym. The logic of bunched implications. *Bulletin of Symbolic Logic*, 5(2), June 1999.

[ORY01] P. W. O'Hearn, J. C. Reynolds, and H. Yang. Local reasoning about programs that alter data structures. In *Proceedings of 15th Annual Conference of the European Association for Computer Science Logic: CSL 2001*, Lecture Notes in Computer Science, Berlin, 2001. Springer-Verlag.

[OYR09] P. O'Hearn, H. Yang, and J. C. Reynolds. Separation and information hiding. *ACM TOPLAS*, 31(3):50 pages, 2009. (Preliminary version appeared in POPL'04, pp268-280).

[Par07] M. Parkinson. Class invariants: the end of the road? Position paper presented at *3rd International Workshop on Aliasing, Confinement and Ownership in Object-oriented Programming*, 2007.

[PB05] M. Parkinson and G. Bierman. Separation logic and abstraction. In *32nd POPL*, pp59–70, 2005.

[Plo73] G. D. Plotkin. Lambda definability and logical relations. Technical Report SAI-RM-4, School of Artificial Intelligence, University of Edinburgh, 1973.

[POY04] D. J. Pym, P. O'Hearn, and H. Yang. Possible worlds and resources: The semantics of BI. *Theoretical Computer Science*, 315(1):257–305, 2004.

[Rey83] J. C. Reynolds. Types, abstraction and parametric polymorphism. *Proceedings of IFIP Congress*, 1983.

[Rey02] J. C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Proceedings of Logic in Computer Science*, volume 17, pages 55 – 74, Copenhagen, July 2002. IEEE.

[Rey05] J. C. Reynolds. Precise, intuitionistic, and supported assertions in separation logic, May 2005. Slides from the invited talk given at the MFPS XXI Available at author's home page: `http://www.cs.cmu.edu/~jcr/`.

[RY03] U. S. Reddy and H. Yang. Correctness of data representations involving heap data structures. In P. Degano, editor, *Proc. of the 12th European Symposium on Programming, ESOP 2003*, pages 223 – 237. Springer Verlag, 2003.

[Sch77] J. Schwarz. Generic commands - a tool for partial correctness formalisms. *Computer Journal*, 10(2):151 – 155, 1977.

[Yan01] H. Yang. *Local Reasoning for Stateful Programs*. PhD thesis, University of Illinois, 2001.

[Yan07] H. Yang. Relational separation logic. *Theoretical Computer Science*, 375(1-3):308–334, 2007. Festschrift for John C. Reynolds.

[YO02] H. Yang and P. O'Hearn. A semantic basis for local reasoning. In *Proceedings of FOSSACS 2002*, 2002.